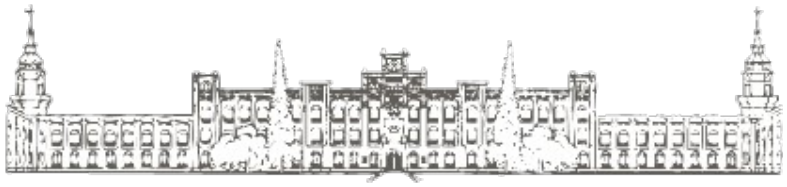

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

КАФЕДРА ІНФОРМАТИКИ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Курсова робота з освітнього компоненту

«Технології паралельних обчислень. Курсова робота»

Тема: **Алгоритм BFS та його паралельна реалізація з використанням мови C++**

Керівник:
проф. Стеценко Інна В’ячеславівна

«Допущено до захисту»
_____

«___» _____ 2024 р.

Виконавець:
Панченко Сергій Віталійович
студент групи ІП-11
залікова книжка № _____
_____

«23» травня 2024 р.

Захищено з оцінкою
_____

Члени комісії:

_____

_____ Антон ДИФУЧИН

Київ – 2024

ЗАВДАННЯ

1. Виконати огляд існуючих реалізацій алгоритму BFS, послідовних та паралельних. Зробити висновок про актуальність дослідження.
2. Виконати розробку послідовного алгоритму BFS використанням мови C++. Дослідити швидкодію алгоритму при зростанні складності обчислень та зробити висновки про необхідність паралельної реалізації алгоритму.
3. Виконати розробку паралельного алгоритму BFS використанням мови C++.
4. Виконати тестування алгоритму, що доводить коректність результатів обчислень.
5. Виконати дослідження швидкодії паралельного алгоритму при зростанні кількості даних для обчислень.
6. Виконати експериментальне дослідження прискорення розробленого алгоритму при зростанні кількості даних для обчислень. Реалізація алгоритму вважається успішною, якщо прискорення більше 1.2.
7. Зробити висновки про переваги паралельної реалізації обчислень для алгоритму, що розглядається у курсовій роботі, та програмних засобів, які використовувались.

АНОТАЦІЯ

Пояснювальна записка до курсової роботи: 73 сторінки, 33 рисунки, 4 таблиці.

Об’єкт дослідження: задача паралельного пошуку шляху у графі.

Мета роботи: теоретично дослідити паралельні методи пошуку шляху у графі за допомогою алгоритму BFS; переглянути відомі їхні реалізації; спроектувати, реалізувати, протестувати послідовний та паралельний алгоритми; дослідити ефективність паралелізації програми;

Виконана програмна реалізація паралельного та послідовного алгоритму пошуку шляху у графі, проведено аналіз їх ефективності.

Ключові слова: ПОШУК ШЛЯХУ У ГРАФІ, BFS, КАНАЛИ, ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ.

ЗМІСТ

ВСТУП

У сучасному інформаційному суспільстві важлива роль відводиться оптимізації алгоритмів для вирішення різноманітних завдань, зокрема, задач пошуку шляхів у графі. Алгоритм пошуку в ширину (BFS) [1] визначається як один із найбільш ефективних та широко застосовуваних для вирішення подібних задач. Використання BFS виявляється актуальним у багатьох сферах, таких як штучний інтелект, робототехніка, комп'ютерні ігри та навігація.

З появою багатоядерних процесорів та розподілених систем виникає потреба в розробці ефективних паралельних алгоритмів, спрямованих на прискорення обчислень. Саме у цьому контексті виникає ідея дослідження можливості паралельної реалізації алгоритму BFS за допомогою мови програмування C++ [2].

Об'єктом даної курсової роботи є вивчення та аналіз підходів до паралельної реалізації алгоритму BFS для графів різних розмірностей. Враховуючи особливості BFS, що базується на результатах попередніх ітерацій, основним завданням є розробка ефективної паралельної реалізації. Також в рамках роботи буде розглянуто та проаналізовано ефективність різних методів паралельного виконання алгоритму з урахуванням його особливостей.

Окрім цього, планується розробка та аналіз нового підходу до паралельної реалізації, який сприяє швидкому знаходженню шляхів у графі, особливо при збільшенні розмірності матриці. Отримані результати і висновки будуть важливим внеском у розуміння проблеми паралельного програмування та оптимізації алгоритмів в сучасних умовах розвитку технологій.

1 ОПИС ПОСЛІДОВНОГО АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ

Алгоритм BFS достатньо нескладно розпалалелити, тому окрім класичної реалізації BFS існує паралельна версія BFS зі спільною пам’яттю, тобто чергою, для всіх потоків. У даній роботі будуть розглянуті реалізації послідовна, паралельна зі спільною чергою, а також власні варіції.

1.1 Послідовний BFS

Алгоритм BFS (Breadth-First Search) є ефективним методом для виявлення шляхів у графі та знаходження найкоротших відстаней від стартової вершини до кінцевої. У початковій вершині встановлюється маркер відвіданої, і сусіди цієї вершини додаються до черги. Процес повторюється, доки не буде відвідано всі вершини графа, або знайдено вершину, що дорівнює кінцевій.

```
Algorithm PredecessorNodesImpl():
    // Ініціалізація
    create queue with start node
    visitorMap = CreateVisitorMap() // Створення мапи відвідувань
    isFoundEndNode = false

    // Основний цикл BFS
    while queue is not empty and not isFoundEndNode:
        currentNode = dequeue front node from queue

        // Перегляд усіх сусідів поточної вершини
        for each neighbour in graph.adjacent(currentNode):
            if not visitorMap[neighbour].visited:
                // Позначаємо сусіда як відвіданого і зберігаємо предка
                visitorMap[neighbour].visited = true
                visitorMap[neighbour].predecessor = currentNode

                // Перевірка, чи є сусід кінцевою вершиною
                if neighbour == end node:
                    isFoundEndNode = true
                    break // Вихід з циклу, якщо знайдено кінцеву вершину

                // Додавання сусіда в чергу
                queue.push(neighbour)
```

```
    // Перевірка, чи знайдено кінцеву вершину
    if not isFoundEndNode:
        return None // Кінцеву вершину не знайдено
    return visitorMap // Повертаємо мапу відвідувань

Algorithm CreateVisitorMap():
    visitorMap = new map
    // Ініціалізація мапи для всіх вершин графа
    for each vertex in graph:
        visitorMap[vertex] = (visited = false, predecessor = None)
    return visitorMap

Algorithm DeterminePath(predecessorNodes):

    // Ініціалізація шляху з кінцевої вершини
    path = [end node]

    // Початкова вершина шляху
    currentNode = path's first element

    // Побудова шляху зворотньо через предків
    while currentNode is not start node:
        // Отримуємо предка поточної вершини
        currentNode = predecessorNodes[currentNode].predecessor
        // Додаємо предка до шляху
        path.add(currentNode)

    // Перевертаємо шлях, щоб він йшов від початку до кінця
    reverse(path)

    // Повертаємо знайдений шлях
    return path
```

`CreateVisitorMap` створює ініціалізує мапу відвідувань, де кожній вершині спочатку присвоюються значення, що вона не відвідана та не має предка.

`PredecessorNodesImpl` виконує пошук в ширину (BFS), де кожна вершина, що відвідується, позначається як відвідана у мапі відвідувань, їй присвоюється предок, і вона додається до черги. Якщо знайдено кінцеву вершину, алгоритм завершується раніше. Якщо кінцеву вершину не знайдено після обходу

всіх вершин, алгоритм повертає `None`.

`DeterminePath` відтворює шлях від початкової до кінцевої вершини, використовуючи мапу предків, яка створюється під час виконання алгоритму пошуку в ширину.

1.2 Паралельний BFS

У паралельній реалізації алгоритму BFS[3], основний потік надсилає певну окрему частину загального фронтиру до дочірнього потоку. Кожен потік проходиться по своїй частині фронтиру, обробляє вершини та додає нові до вихідного фронтиру. Далі відправляє вихідний фронтир назад до основного потоку. Далі основний потік об’єднує результати у загальний фронтир. Цей процес продовжується до тих пір, поки всі вершини графа не будуть відвідані усіма потоками, або знайдено вершину, що дорівнює кінцевій.

Переглянемо псевдокод:

```
// Створює карту для відстеження відвіданих вузлів та їхніх попередників
FUNCTION CreateVisitorMap
    INITIALIZE visitorMap as empty map
    FOR each node in graph
        INSERT node into visitorMap with initial state
    RETURN visitorMap

// Обробляє комунікацію між потоками, включно з відправленням та отриманням повідомлень про стан дослідження вузлів
FUNCTION Communicate(deque, totalEnqueuedNum, visitorMap, senders, listeners)
    INITIALIZE communication result
     HANDLE messages from parent threads and update the deque and visitorMap accordingly
    RETURN communication result

// Виконує частину роботи BFS, засновану на сегменті черги (крадіжка роботи)
FUNCTION DoPartialWork(queueView, visitorMap)
    INITIALIZE partial result
    FOR each node in the segment of the deque
        IF node is unvisited
            MARK node as visited and perform necessary actions
    RETURN partial result
```

```
// Основна функція, яка виконується на кожному дочірньому потоці для паралельного виконання BFS
FUNCTION ChildThreadWork(childSender, parentListener, visitorMap)
    WHILE BFS is not complete
        RECEIVE message from parent
        EXECUTE partial BFS work based on the message
        SEND result back to parent

// Координує ітерацію BFS на декількох потоках
FUNCTION IterateWork(deque, senders, visitorMap)
    INITIALIZE iteration result
    DISTRIBUTE work among child threads and collect results
    UPDATE deque and visitorMap based on children's results
    RETURN iteration result

// Обробляє результати ітерації BFS, перевіряє, чи знайдено кінцевий вузол, чи потрібна додаткова ітерація
FUNCTION ProcessIterationResult(deque, partialResult, senders, totalEnqueued)
    ANALYZE partialResult
    IF end node is found
        RETURN EndNodeFound
    ELSE
        UPDATE deque and continue searching
        RETURN ContinueIteration

// Відправляє повідомлення всім дочірнім потокам
TEMPLATE FUNCTION SendMessageToAll(senders)
    FOR each sender in senders
        SEND message through sender
```

2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ

У рамках даного розділу проводиться детальний огляд процесу розробки послідовного алгоритму BFS. Визначаються основні кроки та етапи, необхідні для створення функціонального та ефективного алгоритмічного рішення. При цьому здійснюється аналіз особливостей графових структур, що може впливати на вибір оптимальних стратегій алгоритму.

Основний акцент розділу робиться на вивченні та порівнянні часових та просторових характеристик розробленого послідовного алгоритму. Це дозволяє визначити його ефективність та потенційні області оптимізації.

2.1 Проектування послідовного алгоритму

Відповідно до теорії, описаної в пунткі 1.1, було розроблено алгоритм BFS. Він використовує хеш-таблицю для зберігання відвіданих вершин, де ключ — вершина, значення — прапор про відвідування; чергу — для зберігання фронтиру — сукупності сусідніх вершин.

2.2 Реалізація послідовного алгоритму

Для відображення графа достатньо використати std::unordered_map, де ключами будуть вершини, значеннями — вектори з сусідніх вершин. Сама вершина — це певний шаблонний тип, що задовольняє стандарний концепт std::regular [4] (тобто тип має реалізовувати конструктор за замовчуванням, копіювання та оператор порівняння), а також має бути спеціалізована функція std::hash [5] для даного типа. На рисунку 2.1 можна побачити вимоги на типи.

```
template<typename T>
concept CBFSUsable = std::regular<T> and requires(T value) {
        {std::hash<T>{}(value)} -> std::same_as<std::size_t>;
    };
```

Рисунок 2.1 Вимоги до шаблонного типа та графа

2.2.1 TBaseBFSMixin

Будь-який клас алгоритму BFS наслідується від шаблонного класа

TBaseBFSMixin, який побудований за принципом CRTP[6] (Curiously Recurring Template Pattern). CRTP дозволяє використовувати статичний поліморфізм, який є більш типізовано безпечним, та реалізовувати поліморфні функції без використання ключового слова virtual, який накладає додаткову ціну на виклик віртуальної функції. На рисунку 2.2 можна побачити оголошення класа TBaseBFSMixin.

```
template<CBFSUsable T, typename Derived>
class TBaseBFSMixin {
    public:
    template<typename... Args>
    static std::optional<std::vector<T>> Do(const AGraph<T>& graph, const T& start, const T& end, Args&&... args);

    protected:
    TBaseBFSMixin(const AGraph<T>& graph, const T& start, const T& end);

    protected:
    std::optional<std::vector<T>> Execute();

    protected:
    const Derived* self() const;
    Derived* self();

    protected:
    template<typename ValueType>
    std::vector<T> DeterminePath(const std::unordered_map<T, ValueType>& predecessorNodes) const;

    protected:
    const AGraph<T>& m_refGraph;
    const T& m_refStart;
    const T& m_refEnd;
};
```

Рисунок 2.2 Оголошення класа TBaseBFSMixin

TBaseBFSMixin побудовний таким чином, що будь-який дочірній алгоритм надає користувачу лише один статичний метод Do, усе інше - приховане, що робить інтерфейс досить простим.

На рисунку 2.3 TBaseBFSMixin неявно вимагає від дочірніх типів реалізувати метод PredecessorNodesImpl, що має опціонально повертати таблицю відвідування.

```
template<CBFSUsable T, typename Derived>
std::optional<std::vector<T>> TBaseBFSMixin<T, Derived>::Execute() {
    if(m_refStart == m_refEnd) return std::vector{m_refStart, m_refEnd};
    const auto result = self()->PredecessorNodesImpl();
    if(not result) return std::nullopt;
    return DeterminePath(result.value());
}
```

Рисунок 2.3 Накладання вимоги реалізувати метод PredecessorNodesImpl

TBaseBFSMixin зберігає в собі посилання на початкову вершину, кінцеву, а також граф. До того ж має метод DeterminePath, який повертає шлях від початкової до кінцевої вершини. На рисунку 2.4 наведена його реалізація.

```
template<CBFSUsable T, typename Derived>
template<typename ValueType>
std::vector<T> TBaseBFSMixin<T, Derived>::DeterminePath(
    const std::unordered_map<T, ValueType>& predecessorNodes) const {
    auto path = std::vector<T>{this->m_refEnd};
    auto currentNode = path.front();
    while(currentNode != this->m_refStart) {
        currentNode = predecessorNodes.at(currentNode).second;
        path.push_back(currentNode);
    }
    std::reverse(path.begin(), path.end());
    return path;
}
```

Рисунок 2.4 Реалізація метода DeterminePath

2.2.2 TSequentialBFS

Послідовний алгоритм BFS реалізований у вигляді дочірнього класа TBaseBFSMixin TSequentialBFS. На рисунку 2.5 та 2.6 відповідно він визначає метод CreateVisitorMap, що створює хеш-таблицю з ключами-вершинами та значеннями — прапорами відвідування, та реалізовує метод PredecessorNodesImpl.

```
template<CBFSUsable T>
TSequentialBFS<T>::AVisitorMap TSequentialBFS<T>::CreateVisitorMap() const {
    auto visitorMap = std::unordered_map<T, std::pair<bool, T>>();
    visitorMap.reserve(this->m_refGraph.size());
    for(const auto& [key, _] : this->m_refGraph) {
        visitorMap.insert_or_assign(key, std::make_pair(false, T()));
    }
    return visitorMap;
}
```

Рисунок 2.5 Визначення метода CreateVisitorMap

У методі PredecessorNodesImpl спочатку створюється черга, або фронтир вершин, таблиця відвідування; у циклі, доки черга не пуста, або не знайдена кінцева вершина, алгоритм обходить кожну вершину фронтира, позначає її обійденою, дивиться її сусідів, додає їх у чергу, якщо вони не були раніше відвідані, повторює

цю операцію знову.

```
template<CBFSUsable T>
std::optional<typename TSequentialBFS<T>::AVisitorMap>
    TSequentialBFS<T>::PredecessorNodesImpl() const {
    auto queue = std::queue<T>({this->m_refStart});
    auto visitorMap = CreateVisitorMap();
    auto isFoundEndNode = false;
    while(not queue.empty() and not isFoundEndNode) {
        const auto currentNode = std::move(queue.front());
        queue.pop();
        for(const auto& neighbour : this->m_refGraph.at(currentNode)) {
            const auto neighbourIt = visitorMap.find(neighbour);
            if(not neighbourIt->second.first) {
                neighbourIt->second.first = true;
                neighbourIt->second.second = currentNode;
                if(neighbour == this->m_refEnd) {
                    isFoundEndNode = true;
                    break;
                }
                queue.push(neighbour);
            }
        }
    }
    if(not isFoundEndNode) return std::nullopt;
    return visitorMap;
}
```

Рисунок 2.6 Реалізація метода PredecessorNodesImpl

2.3 Тестування послідовного алгоритму

Тестування усіх алгоритмів проводиться за допомогою бібліотеки googletest[7], яка є стандартом тестування для багатьох C++ проектів. Бібліотека надає класи, методи, макроси тестування, щоб полегшити та зробити більш зрозумілим тестування.

Для початку потрібно визначити метод створення графа. На рисунку 2.7 можна побачити, що граф — це квадрат N*N вершин, де кожна вершина знає про найближчі вершини по горизонталі, вертикалі та діагоналях.

На рисунку 2.8 можна побачити визначення метода Create2DGrid. У циклі з індекса визначається координата вершини, далі до координати вершини додаються

числа -1, 0, 1, потім перевіряється, чи утворена вершина не виходить за межі сітки.

Коректність шляху перевіряється методом IsPathValid, який перевіряє, чи кожна вершина знаходиться в графі, чи кожний наступний елепент шляху є сусідом поточної вершини. На рисунку 2.9 можна побачити визначення даного методу.

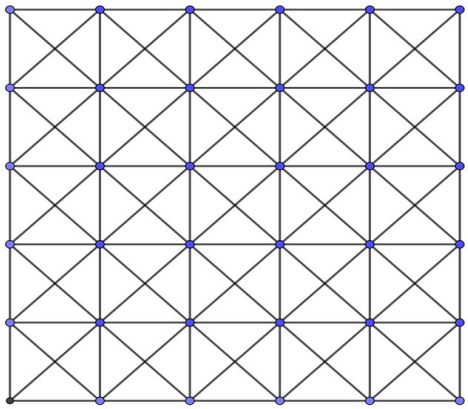

Рисунок 2.7 Вигляд графа для тестування

```
std::unordered_map<unsigned, std::vector<unsigned>>
TTestBFSFixture::Create2DGrid(const unsigned int size) {
    auto grid = std::unordered_map<unsigned, std::vector<unsigned>>();
    const auto totalSize = size * size;
    grid.reserve(totalSize);
    for(auto index = 0u; index < totalSize; ++index) {
        const auto x = static_cast<int>(index % size);
        const auto y = static_cast<int>(index / size);
        const auto utmost = static_cast<int>(size) - 1;
        auto neighbourIndexes = std::vector<unsigned>();
        for(auto deltaY = -1; deltaY <= 1; ++deltaY) {
            const auto newY = y + deltaY;
            if(newY < 0 or newY > utmost) continue;
            const auto base = static_cast<unsigned>(newY) * size;
            for(auto deltaX = -1; deltaX <= 1; ++deltaX) {
                if(deltaY == 0 && deltaX == 0) continue;
                const auto newX = x + deltaX;
                if(newX < 0 or newX > utmost) continue;
                const auto offset = static_cast<unsigned>(newX);
                neighbourIndexes.push_back(base + offset);
            }
        }
        grid.insert_or_assign(index, neighbourIndexes);
    }
    return grid;
}
```

Рисунок 2.8 Визначення метода Create2DGrid

```cpp
bool TTestBFSFixture::IsPathValid(
    const std::vector<unsigned int>& path,
    const bfs::AGraph<unsigned int>& graph) {
    for(const auto& [start, end] : path | std::views::pairwise) {
        const auto it = graph.find(start);
        if(it == graph.end()) {
            return false;
        }
        const auto isContain = std::ranges::contains(it->second, end);
        if(not isContain) {
            return false;
        }
    }
    return true;
}
```

Рисунок 2.9 Визначення метода IsPathValid

На рисунку 2.10 можна побачити, що проводиться пошук шляху від вершини з індексом 0, що знаходиться у верхньому лівому куті графа, до вершини lastIndex, що знаходиться у нижньому правому куті графа. Далі вимірюється час у мілсекундах та перевіряється, чи справді створений алгоритмом шлях валідний. Усі результати вимірювання записуються у JSON-форматі [8], де вказується назва алгоритму, кількість елементів у графі, кількість мілісекунд виконання, щоб надалі з допомогою Python можна було побудувати графіки порівняння алгоритмів.

```cpp
const auto [sequentialMillis :const double , singleRes :const vector<unsigned> ] = [&grid, &lastIndex, &size]() ->tuple<double, vector<unsigned>> {
    const auto start :time_point<system_clock> = std::chrono::system_clock::now();
    auto result :optional<vector<unsigned>> = bfs::TSequentialBFS<unsigned>::Do(grid, start: 0, end: lastIndex);
    const auto delay :common_type<...>::type = std::chrono::system_clock::now() - start;
    const auto millis :long = std::chrono::duration_cast<std::chrono::milliseconds>(delay).count();
    EXPECT_TRUE(IsPathValid(result.value(), grid));
    WriteToReport( str: std::format( fmt: "{{ \"name\": {}, \"size\": {}, \"milliseconds\": {} }}", "Sequential", size, millis));
    return std::make_tuple(static_cast<double>(millis), std::move(result.value()));
}();
```

Рисунок 2.10 Тестування послідовного алгоритму

У таблиці 2.1 наведено залежність часу виконання від кількості елементів у графі. Для кращої репрезентації результатів побудуємо графік, який наведений на рисунку 2.11. Як бачимо, зі збільшенням елементів час також зрозстає.

Тестування алгоритму відбувалуся на пристрої на рисунку 2.11 за умов постійного живлення, відключеного з’єднання з мережею, відсутності роботи сторонніх програм, консольного інтерфейсу.

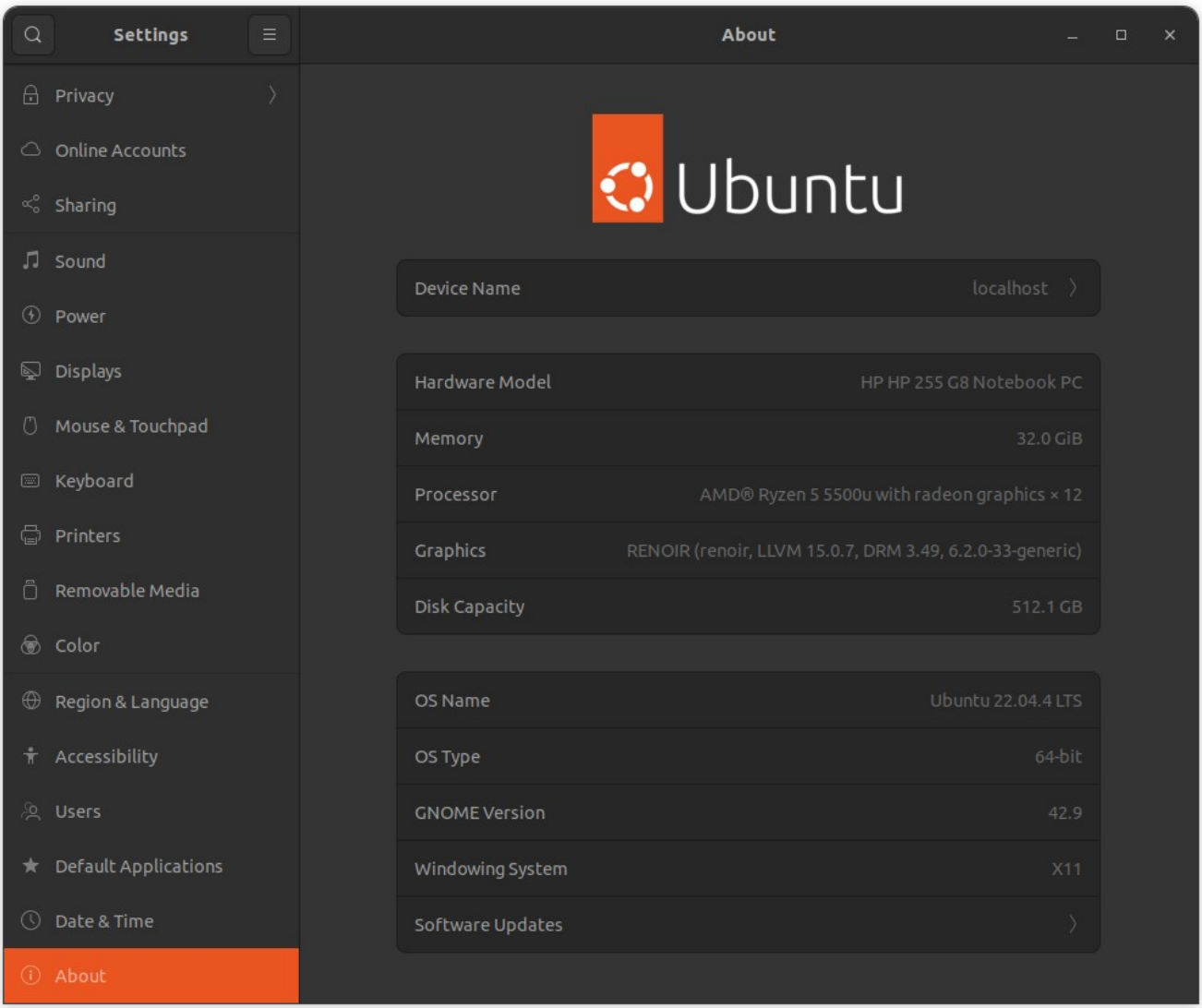

Рисунок 2.11 Характеристики ноутбука

Таблиця 2.1 Тестування послідовного алгоритму

Кількість вершин	Час виконання у мілісекундах
6250000	977
6890625	1162
7562500	1273
8265625	1412
9000000	1534
9765625	1872
10562500	2059
12250000	2653
13213225	3137
14062500	3595
15015625	3659

16000000	3809

На рисунку 2.12 можна побачити графік залежності

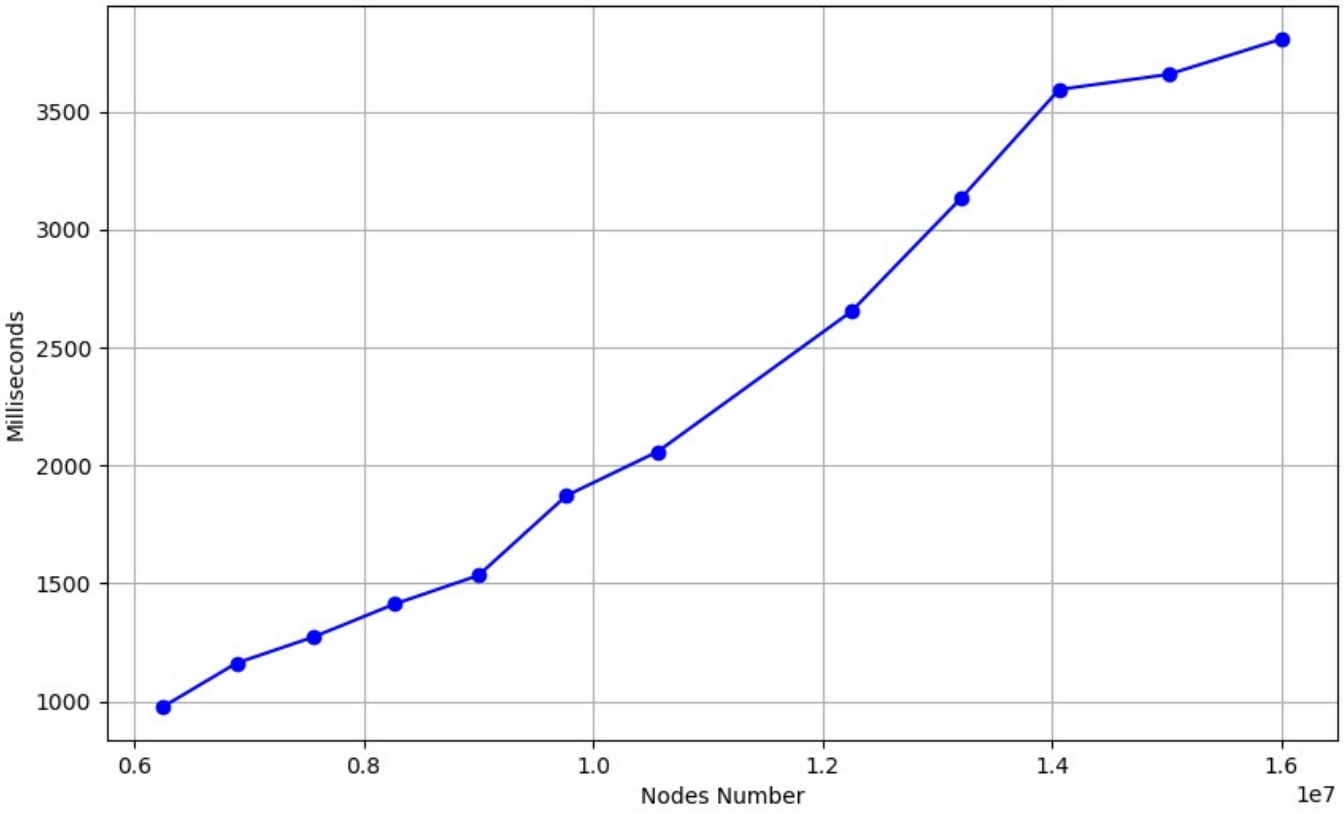

Рисунок 2.12 Графік залежності часу від розміру графа для послідовного алгоритму

Бачимо, що різниця у часі для графів з розмірами 2500 та 4000 відрізняється ледь не втричі, тому є сенс розпалалелити алгоритм.

2.4 Висновок

У даному розділі було проведено детально розробку алгоритму BFS на основі класу TSequentialBFS. Також описали процеси тестування: створення графа, вимірювання часу виконання алгоритму від розміру графів. Результати виконання записуються у JSON-форматі для зручної обробки результатів.

Як побачили час виконання помітно зростає зі збільшенням розміру графа, тому потенціал до покращення з допомогою паралелезації існує.

3 ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС

Для реалізації паралельного алгоритму BFS я обрав C++. Вона є потужною системною мовою програмування, яка надає високі абстракції над низькорівневими речами без втрати швидкості.

C++ включає в стандартну бібліотеку засоби паралельного програмування, як-от: потоки, м’ютекси, атомарні типи тощо. Для цього він має відповідні файли <thread>[9], <mutex>[10], <atomic>[11] тощо.

Також варто розглянути засоби, які існують, але не будуть використані у даній роботі.

Boost.MPI [12] — це файл бібліотеки Boost, що надає ООП обгорти для MPI API мови C. Недоліком даного засобу є те, що він вимагає нестандартний компілятор MPI, який не підтримує нові версії мови C++.

Boost.Asio [13] — це файл бібліотеки Boost, що відповідає за асинхронний вивід та ввід даних у мережі. Цю бібліотеку можна було б використатити для передавання повідомлень між потоками, але, на жаль, Boost.Asio надає можливість створювати канали між потоками тільки у вигляді сокетів та передавати дані тільки як байти без прив’язки до типів.

4 РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЕКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ

У рамках даного розділу проводиться детальний огляд процесу розробки паралельного алгоритму BFS. Як і в розділі 2 визначаються основні кроки та етапи, необхідні для створення функціонального та ефективного алгоритмічного рішення. Детально розглядаються можливість розпаралелювання, вибір структур даних та оптимізаційні підходи, що можуть покращити продуктивність алгоритму. Також буде проведений аналіз результатів тестування та визначені потенційні області оптимізації для забезпечення оптимальної швидкодії паралельного алгоритму BFS.

4.1 Структури даних

Найпростішим варіатом розпалалелити алгоритм є зробити потокостійкий клас таблиці відвідування розділити її між всіма потоками. Однак у такої реалізації є суттєві проблеми.

Розглянемо для початку розділення таблиці відвідування між всіма потоками. Логічно утворити окремий клас TThreadSafeMap, у якому буде м’ютекс та таблиця відвідування. Таким чином буде забезпечено, що тільки один потік буде модифікувати таблицю в певний момент. На рисунку 4.1 зображено схему даної реалізації.

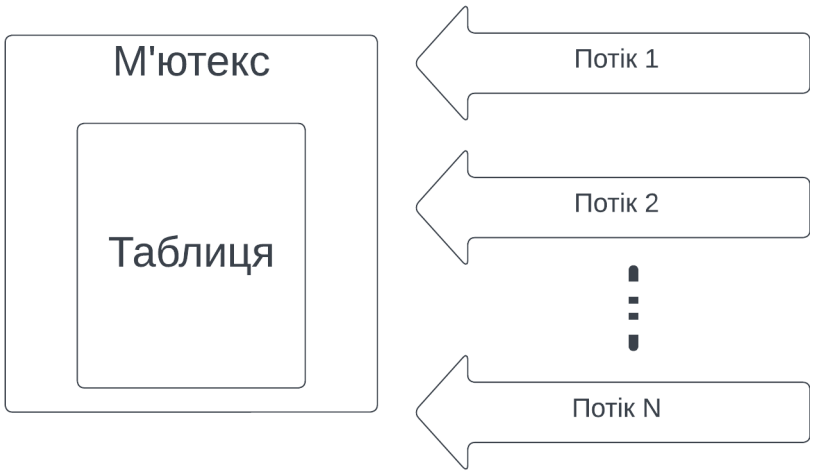

Рисунок 4.1 Таблиця відвідування покрита суцільним м’ютексом

Однак можна зрозуміти, що таблиця буде створена один раз на початку

алгориму, і її розмір не буде змінюватися, лише її елементи будуть приймати інші значення. Поглянемо на популярні бази даних, як-от: PostgreSQL[14], MS SQL[15] — усі вони вирішують дану проблему надаючи м’ютекс лише на певну частину даних. Тому набагато ефективніше буде мати в кожному записі атомарний прапор, який буде позначати, чи була вершина раніше відвідана чи ні. На рисунку 4.2 можна побачити схему даної реалізації.

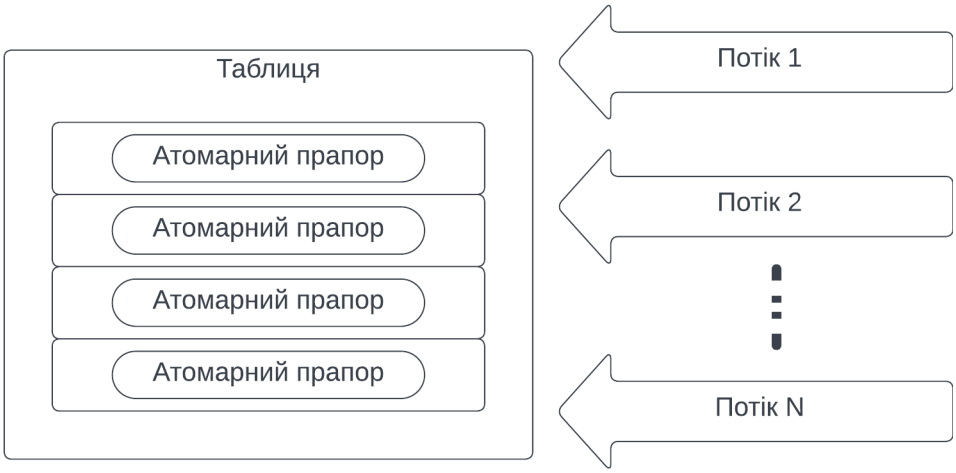

Рисунок 4.2 Таблиця відвідування з атомарними прапорами

Основний потік буде через певний канал спілкуватися з дочірнім потоком та передавати повідомлення про виконання алгоритму. Таким чином буде один раз створено дочірні потоки, їм будуть надані канали спілкування з основним потоком; кожен дочірній потік буде обробляти лише свою частину черги, заповнювати власну локальну чергу, передавати її назад основному потоку; потім основний потік збирає докупи результати виконання від усіх потоків, далі надсилає інформацію про те, яку саму частку спільної черги обробити; далі цей алгоритм продовжується, допоки не будуть оброблені всі вершини, або знайдена кінцева.

На рисунку 4.3 показана загальна ідея алгоритму.

Отже, написання поточної версії алгоритму складається з декількох пунктів:

— написати певну абстракцію, яку назвемо “канал”, через яку будуть передаватися певні типи;
— написати певний контейнер, який буде зберігати сукупність векторів, тобто результи дочірніх потоків; даний контейнер має ззовні здаватися неперервним шматком даних, до яких можна дістатися за допомогою ітераторів;
— написати загальну схему алгоритму в псевдокоді.

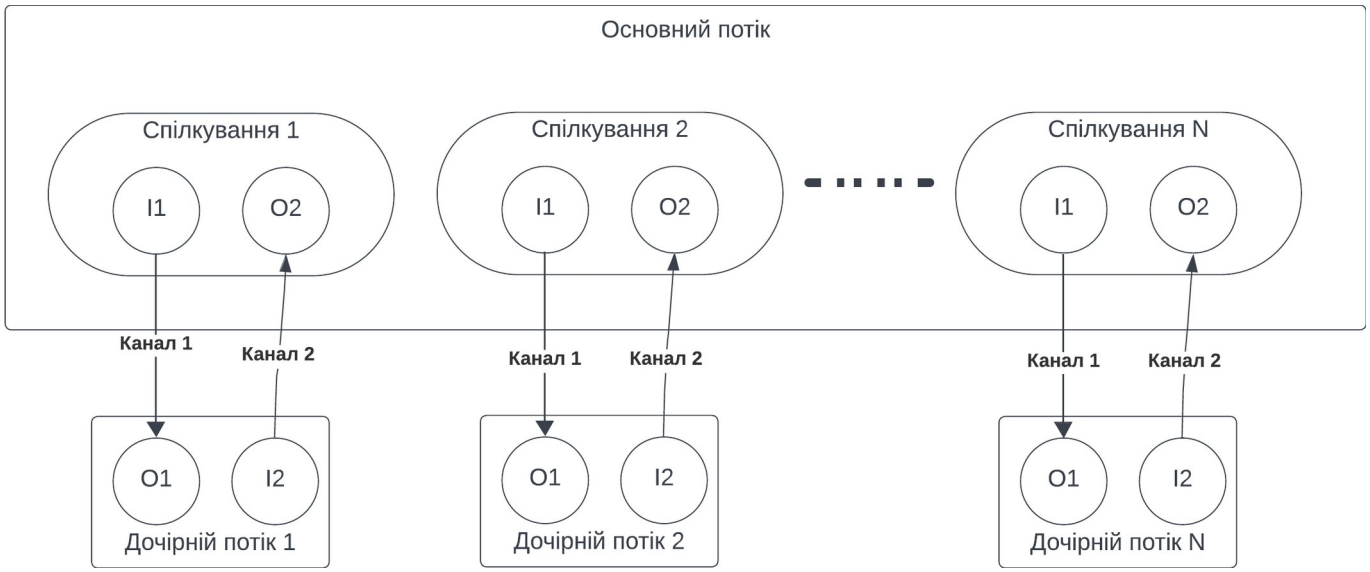

Рисунок 4.3 Загальна ідея BFS з повідомленнями

4.2 Проектування та реалізація паралельних алгоритмів

У даному підрозділі будуть розглянуті реалізації класів, що використовуються в паралельних версіях алгоритмів. За потреби будуть зображені схеми для кращого розуміння.

4.2.1 TPipeReader, TPipeWriter, TPipeChannel

TPipeChannel — класи, що відповідають за реалізацію каналу передачі даних між потоками. Він складається з входу — TPipeWriter — та виходу — TPipeReader. Обидва кінця каналу володіють посиланнями на дані, що передаються. Самі дані захищені атомарним прапором. Обидва кінці мають методи Write та Read, які очікують виконання доки дані не будуть звільненні на вписування або на зчитування. Для кращого розуміння роботи зображено рисунок 4.7.

Розглянемо C++ реалізацію даної схеми. TPipeChannel — це клас, що зберігає в собі обидва кінця канала, у конструкторі він надає каналам спільне посилання на пару з атомарного прапора та даних. Дані мають реалізовувати конструктори за замовчуванням і переміщення. На рисунку 4.8 позначені обмеження. На рисунку 4.9 показується реалізація класа TPipeChannel.

TPipeWriter та TPipeReader мають лише по одному публічному методу Write та Read і конструктор переміщення, інші конструктори, методи, поля класів інкапсульовані. Обидва класи знають, що їхнім дружнім типом є TPipeChannel, і

тільки він може створювати об’єкти даних класів. Це зроблено для того, щоб не можна було окремо створити чи скопіювати або TPipeWriter або TPipeReader, вони існують лише в парі. Загалом класи однакові за будовою, тому буде показано лише оголошення класу TPipeReader на рисунку 4.10.

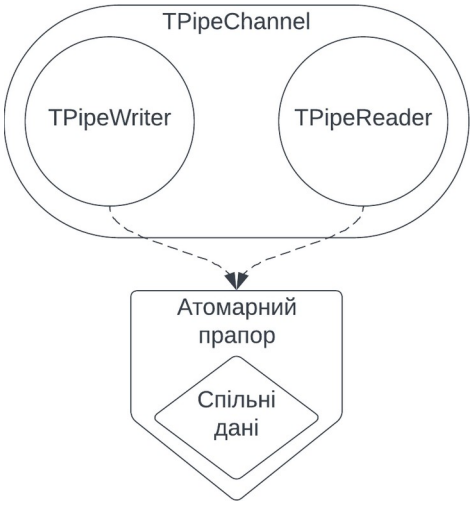

Рисунок 4.4 Схема роботи каналу

```
template<typename T>
concept CPipeUsable = std::default_initializable<T> and std::movable<T>;
```

Рисунок 4.5 Обмеження на тип в каналах

```
template<CPipeUsable T>
class TPipeChannel {
    public:
    TPipeChannel();

    public:
    TPipeWriter<T> Writer;
    TPipeReader<T> Reader;
};

template<CPipeUsable T>
TPipeChannel<T>::TPipeChannel() {
    auto data = std::make_shared<std::pair<T, std::atomic_flag>>();
    Writer = TPipeWriter(data);
    Reader = TPipeReader(std::move(data));
}
```

Рисунок 4.6 Визначення класу TPipeChannel

```
template<CPipeUsable T>
class TPipeReader {
    friend class TPipeChannel<T>;

    public:
    ~TPipeReader()=default;
    TPipeReader(TPipeReader&& other) noexcept;
    TPipeReader& operator=(TPipeReader&& other) noexcept;

    public:
    T Read() const;

    protected:
    TPipeReader() = default;
    TPipeReader(const std::shared_ptr<std::pair<T, std::atomic_flag>>& data);
    TPipeReader(const TPipeReader&) = delete;
    TPipeReader& operator=(const TPipeReader&) = delete;

    protected:
    std::shared_ptr<std::pair<T, std::atomic_flag>> m_pData = nullptr;
};
```

Рисунок 4.7 Оголошення класу TPipeReader

4.2.2 TDeque

TDeque — клас, що виконує роль представлення для вектора векторів. Особливістю цього класу є те, що він дозволяє користувачу ітеруватися по ньому, як неперервному контейнеру. Для цього клас має публічний метод Loop, який приймає індекси початку та кінця, а також лямбду, до якої будуть передаватися елементи під час ітерації. На рисунку 4.11 зображений метод Loop.

Як бачимо, беруться ітератори внутрішніх векторів, потім як тільки досягається кінець внутрішнього ітератора, зовнішній йде до наступного вектора.

```cpp
template<typename T>
void TDeque<T>::Loop(const size_t begin, const size_t end,
    const std::function<void(const T&)>& func) const {
    if(begin == end) return;
    auto vectorIt = m_vData.begin();
    auto elIt = m_vData.begin()->begin();

    auto delay = begin;
    auto dist = vectorIt->end() - elIt;
    while(delay >= dist) {
        delay -= dist;
        ++vectorIt;
        elIt = vectorIt->begin();
        dist = vectorIt->end() - elIt;
    }

    elIt += delay;

    for(auto i = begin; i < end; ++i, ++elIt) {
        if(elIt == vectorIt->end()) {
            ++vectorIt;
            elIt = vectorIt->begin();
        }
        func(*elIt);
    }
}
```

Рисунок 4.8 Визначення методу Loop

4.2.3 TCommunicationBFS

TCommunicationBFS — це паралельна реалізація BFS з повідомленнями. Розглянемо структуру даного класу.

Для початку розглянемо повідомлення, з допомогою яких потоки будуть спілкуватися між собою. Повідомлення відображені у вигляді структур. На рисунку 4.9 можна побачити реалізацію.

Як бачимо, повідомлення включають:

— SEndNodeFound — знайдена кінцева вершина;

— ScontinueIteration — продовження роботи алгоритму;

— SAllNodesEnqueued — усі вершини обійдені;

— SQueueView — певна частина фронтиру вершин, які має обійти дочірній потік;

— SFrontier — частина загального фронтиру, утворена дочірнім потоком.

Повідомлення об’єднані у два варіативних типи на рисунку 4.10:

— AParentMessage — повідомлення від основного потоку до дочірніх;

— AChildrenMessage — повідомлення від дочірніх потоків до основного.

— ACommunicationResult — повідомлення про успішність виконання алгоритму;

— AIterationResult — повідомлення про успішність виконання ітерації алгоритму;

```
protected:
// Messages
struct SContinueIteration {};
struct SEndNodeFound {};
struct SAllNodesEnqueued {};
struct SQueueView {
    const TDeque<T>* Deque;
    size_t Begin;
    size_t End;
};
struct SFrontier {
    std::vector<T> Data;
};
```

Рисунок 4.9 Типи повідомлень

Розглянемо метод PredecessorNodesImpl на рисунку 4.11. У ній ми ініціалізуємо спільний дек, у який зберігається фронтир, мапу відвідування, відправників батьківських повідомлень та читачів дочірніх повідомлень. Далі всередині викликається метод Communicate.

Усередині метода Communicate на рисунку 4.12 ініціалізуються дочірні потоки. Далі у циклі викликаємо IterateWork та очікуємо результати дочірніх потоків. Якщо хтось з них знайшов кінцеву вершину, то виходимо із циклу, якщо її так і не знайдено, то повертаємо відповідне повідомлення.

```cpp
using AParentMessage = std::variant<
    SEndNodeFound,
    SAllNodesEnqueued,
    SQueueView>;

using AChildrenMessage = std::variant<
    SEndNodeFound,
    SFrontier>;

using ACommunicationResult = std::variant<
    SAllNodesEnqueued,
    SEndNodeFound
>;

using AIterationResult = std::variant<
    SEndNodeFound,
    SContinueIteration
>;
```

Рисунок 4.10 Варіативні типи повідомлень

```cpp
template<CBFSUsable T>
std::optional<typename TCommunicationBFS<T>::AVisitorMap>
    TCommunicationBFS<T>::PredecessorNodesImpl() const {

    auto deque = TDeque<T>();
    deque.Push( value: {this->m_refStart});
    size_t totalEnqueuedNum = 0;
    auto visitorMap = this->CreateVisitorMap();
    auto senders = std::vector<TPipeWriter<AParentMessage>>();
    auto listeners = std::vector<TPipeReader<AChildrenMessage>>();

    const auto result :variant<...> = Communicate([&]deque, [&]totalEnqueuedNum,
        [&]visitorMap, [&]senders, [&]listeners);

    switch(result.index()) {
        case VariantIndex<ACommunicationResult, SAllNodesEnqueued>(): {
            return std::nullopt;
        }
        case VariantIndex<ACommunicationResult, SEndNodeFound>(): {
            return visitorMap;
        }
        default: {
            lr::UnsopportedCaseError();
        }
    }
}
```

Рисунок 4.11 Визначення PredecessorNodesImpl

```cpp
template<CBFSUsable T>
auto TCommunicationBFS<T>::Communicate(
        TDeque<T>& deque,
        size_t& totalEnqueuedNum,
        typename TCommunicationBFS::AVisitorMap& visitorMap,
        std::vector<TPipeWriter<AParentMessage>>& senders,
        std::vector<TPipeReader<AChildrenMessage>>& listeners
    ) const -> ACommunicationResult {

    auto threads = std::vector<std::jthread>();
    for(auto i = 0u; i < this->m_uThreadsNum - 1; ++i) {
        auto [parentSender:TPipeWriter<...>, parentListener:TPipeReader<...>] = TPipeChannel<AParentMessage>();
        auto [childrenSender:TPipeWriter<...>, childrenListener:TPipeReader<...>] = TPipeChannel<AChildrenMessage>();
        senders.push_back(x:std::move(parentSender));
        listeners.push_back(x:std::move(childrenListener));
        threads.emplace_back([this,
            sender=std::move(childrenSender),
            listener=std::move(parentListener), &visitorMap]->void {
            ChildThreadWork(sender, listener, [&]visitorMap);
        });
    }

    while(true) {
        auto newDeque = TDeque<T>();
        {
            auto partRes:variant<...> = IterateWork(deque, senders, [&]visitorMap);
            const auto iterResult:variant<...> = ProcessIterationResult([&]newDeque,
                partialResult:std::move(partRes), senders, [&]totalEnqueuedNum);
            if(std::holds_alternative<SEndNodeFound>(iterResult)) {
                return SEndNodeFound{};
            }
        }
        for(auto& l:TPipeReader<...>& : listeners) {
            auto partRes:variant<...> = l.Read();
            auto iterResult:variant<...> = ProcessIterationResult([&]newDeque,
                partialResult:std::move(partRes), senders, [&]totalEnqueuedNum);
            if(std::holds_alternative<SEndNodeFound>(iterResult)) {
                return SEndNodeFound{};
            }
        }
        deque = std::move(newDeque);
        if(totalEnqueuedNum >= this->m_refGraph.size()) {
            return SendMessageToAll<SAllNodesEnqueued>(senders);
        }
    }

    return SAllNodesEnqueued{};
}
```

Рисунок 4.12 Визначення метода Communicate

У методі IterateWork на рисунку 4.13 з між потоками рівномірно розподіляється загальний фронтир.

```cpp
template<CBFSUsable T>
auto TCommunicationBFS<T>::IterateWork(
    const TDeque<T>& deque,
    const std::vector<TPipeWriter<AParentMessage>>& senders,
    typename TCommunicationBFS::AVisitorMap& visitorMap
) const -> AChildrenMessage {
    const auto dequeSize :size_t = deque.Size();
    const auto step :unsigned long = dequeSize / this->m_uThreadsNum;
    const auto remainder :unsigned long = dequeSize % this->m_uThreadsNum;
    for(size_t t = 0, index = 0; t < this->m_uThreadsNum; ++t) {
        auto queueView = SQueueView{};
        queueView.Deque = &deque;
        if(index >= dequeSize) {
            queueView.Begin = dequeSize;
            queueView.End = dequeSize;
        } else {
            const auto localStep :unsigned long = t < remainder ? step + 1 : step;
            queueView.Begin = index;
            queueView.End = index + localStep;
            index += localStep;
        }
        if(t == this->m_uThreadsNum - 1) {
            return DoPartialWork(queueView, [&]visitorMap);
        } else {
            senders[t].Write(queueView);
        }
    }
    return SEndNodeFound{};
}
```

Рисунок 4.13 Визначення метода IterateWork

Розглянемо метод DoPartialWork на рисунку 4.14, де у кожному потоці при обході своєї частини фронтиру, беруться сусіди поточної вершини та відбераються до результуючого фронтиру тільки ті, які не були відвідані на даний момент. Якщо вершина не відвідана, то її атомарний прапор позначається, та перевіряється, чи вона не є кінцевою. Якщо знайдена кінцева вершина, то повертається відповідне повідомлення, інакше — повертається фронтир.

```
template<CBFSUsable T>
auto TCommunicationBFS<T>::DoPartialWork(
    const SQueueView& queueView,
    typename TCommunicationBFS::AVisitorMap& visitorMap
) const -> AChildrenMessage {

    auto frontier = SFrontier();
    auto isEndNodeFound = false;
    queueView.Deque->Loop( begin: queueView.Begin, end: queueView.End,
        func: [this, &frontier, &isEndNodeFound, &visitorMap](const T& node) ->void {
        for(const auto& neighbour : this->m_refGraph.at(node)) {
            const auto neighbourIt = visitorMap.find(neighbour);
            if(neighbourIt->second.first.test_and_set())
                continue;
            neighbourIt->second.second = node;
            if(neighbour == this->m_refEnd) {
                isEndNodeFound = true;
                return;
            }
            frontier.Data.push_back(neighbour);
        }
    });

    if(isEndNodeFound) return SEndNodeFound{};

    return frontier;
}
```

Рисунок 4.14 Визначення метода DoPartialWork

4.3 Тестування алгоритму

У даному підрозділі буде проведено тестування не тільки паралельного алгоритму, а й структур даних, що допомогли їх реалізувати. Умови аналогічні описаним у пункті 2.3.

Проведемо за таблицею 4.1 тестування TPipeChannel, TPipeWriter, TPipeReader. На рисунку 4.15 та 4.16 зображено код тесту та результат відповідно.

Таблиця 4.1 Тестування TPipeChannel, TPipeWriter, TPipeReader

Тест	Перевірка роботи TPipeChannel, TPipeWriter, TPipeReader
Номер тесту	1
Початковий стан	Маємо вхід і вихід каналу
Вхідні дані	Число 10
Опис проведення тесту	Створимо два потоки, змусимо перший потік зупинитися на 4 секунди.
Очікуваний результат	Другий буде очікувати отримання даних, тобто числа 10.
Фактичний результат	Число 10 вивелося на екрані.

```
TEST(Pipes, Transfer) {
    auto [w, r] = bfs::TPipeChannel<int>();

    auto sender = std::jthread([ww=std::move(w)]() {
        using namespace std::chrono_literals;
        std::this_thread::sleep_for(4s);
        ww.Write(10);
    });

    auto listener = std::jthread([rr=std::move(r)]() {
        EXPECT_EQ(rr.Read(), 10);
    });
}
```

Рисунок 4.15 Тест працездатності TPipeChannel, TPipeWriter, TPipeReader

Рисунок 4.16 Успішне проходження тестування TPipeChannel, TPipeWriter, TPipeReader

Проведемо тестування за таблицею 4.2 TDeque. На рисунку 4.17 а 4.18 показано результати проведення та код тесту.

Таблиця 4.2 Тестування TDeque

Тест	Перевірка роботи TDeque
Номер тесту	2
Початковий стан	Маємо TDeque заповнений декількома векторами
Вхідні дані	Пари початкових та кінцевих індексів: {0, 15}, {3, 12}

Опис проведення тесту	Викликаємо метод Loop, передаємо початкові та кінцеві ідекси.
Очікуваний результат	Перевіряємо, чи справді початкові та кінцеві індекси в кінці будуть рівні.
Фактичний результат	Вони рівні.

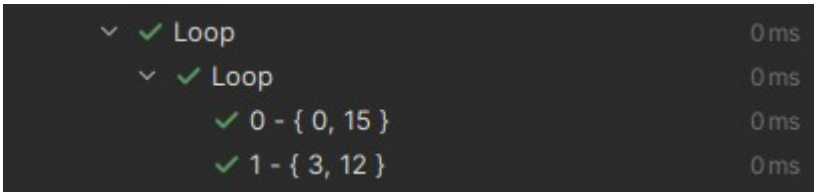

Рисунок 4.17 Успішне проходження тестування TDeque

```
class TDequeTest
    : public testing::TestWithParam<std::array<int, 2>> {
    protected:
    static void SetUpTestSuite();
    static bfs::TDeque<int> s_vDeque;
};

void TDequeTest::SetUpTestSuite() {
    s_vDeque.Push({0, 1, 2, 3, 4});
    s_vDeque.Push({5, 6, 7, 8, 9});
    s_vDeque.Push({10, 11, 12, 13, 14});
}

bfs::TDeque<int> TDequeTest::s_vDeque = bfs::TDeque<int>();

INSTANTIATE_TEST_SUITE_P(Loop, TDequeTest,
    testing::Values(std::array{0, 15}, std::array{3, 12}));

TEST_P(TDequeTest, Loop) {
    auto [beginIt, endIt] = GetParam();
    s_vDeque.Loop(beginIt, endIt, [&beginIt](const auto& el) {
        EXPECT_EQ(el, beginIt);
        ++beginIt;
    });
    EXPECT_EQ(beginIt, endIt);
}
```

Рисунок 4.18 Тест працездатності TDeque

Аналогічно до пункту 2.3 проводемо тестування TCommunicationBFS на коректність утвореного шляху за таблицею 4.3. Код тестування можна переглянути на рисунку 4.19.

Таблиця 4.3 Тестування алгоритмів TCommunicationBFS

Тест	Перевірка роботи TCommunicationBFS
Номер тесту	3
Початковий стан	Маємо початкові дані
Вхідні дані	Кількість потоків: 2, 3, 4, 5, 6, 7, 8, 9 Розмір графів: 200, 300
Опис проведення тесту	У циклі для кожної кількості потоку, для кожного розміру графа пееревіряємо чи повертається коректний шлях, та чи цей шлях співпадає з тим, що повернув однопотоковий алгоритм.
Очікуваний результат	Шлях існує та дорівнює однопотоковій реалізації.
Фактичний результат	Шлях існує та дорівнює однопотоковій реалізації.

```
for(const auto threadsNum :unsigned : threadsNums) {
    const auto start :time_point<system_clock> = std::chrono::system_clock::now();
    const auto result :optional<vector<unsigned>> = bfs::TCommunicationBFS<unsigned>::Do(grid, start: 0, end: lastIndex, threadsNum);
    const auto delay :common_type<...>::type = std::chrono::system_clock::now() - start;
    const auto millis = static_cast<double>(std::chrono::duration_cast<std::chrono::milliseconds>(delay).count());
    EXPECT_TRUE(IsPathValid(result.value(), grid));
    EXPECT_EQ(result.value(), singleRes);
    WriteToReport( str: std::format( fmt: "{{ \"name\": \"{}\", \"size\": {}, \"threadsNum\": {}, \"milliseconds\": {}, \"acceleration\": {} }}",
        "Communication", size, threadsNum, millis, sequentialMillis / millis));
}
```

Рисунок 4.19 Тестування коректності TCommunicationBFS

5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМІВ

Для усередненої оцінки ефективності алгоритмів для кожної комбінації параметрів усі тести було провдено 5 разів. Усі тести мали однаковий початок у лівому верхньому куті графа та кінець у правому нижньому куті графа. Тести проводилися на графах достатньо великого розміру, щоб можна було помітити суттєву різницю.

У додатку А можна перешлянути результати детальніше.

На рисунках 5.1, 5.2 показано залежність часу від розміру графа, прискорення від розміру графа при різній кількості потоків.

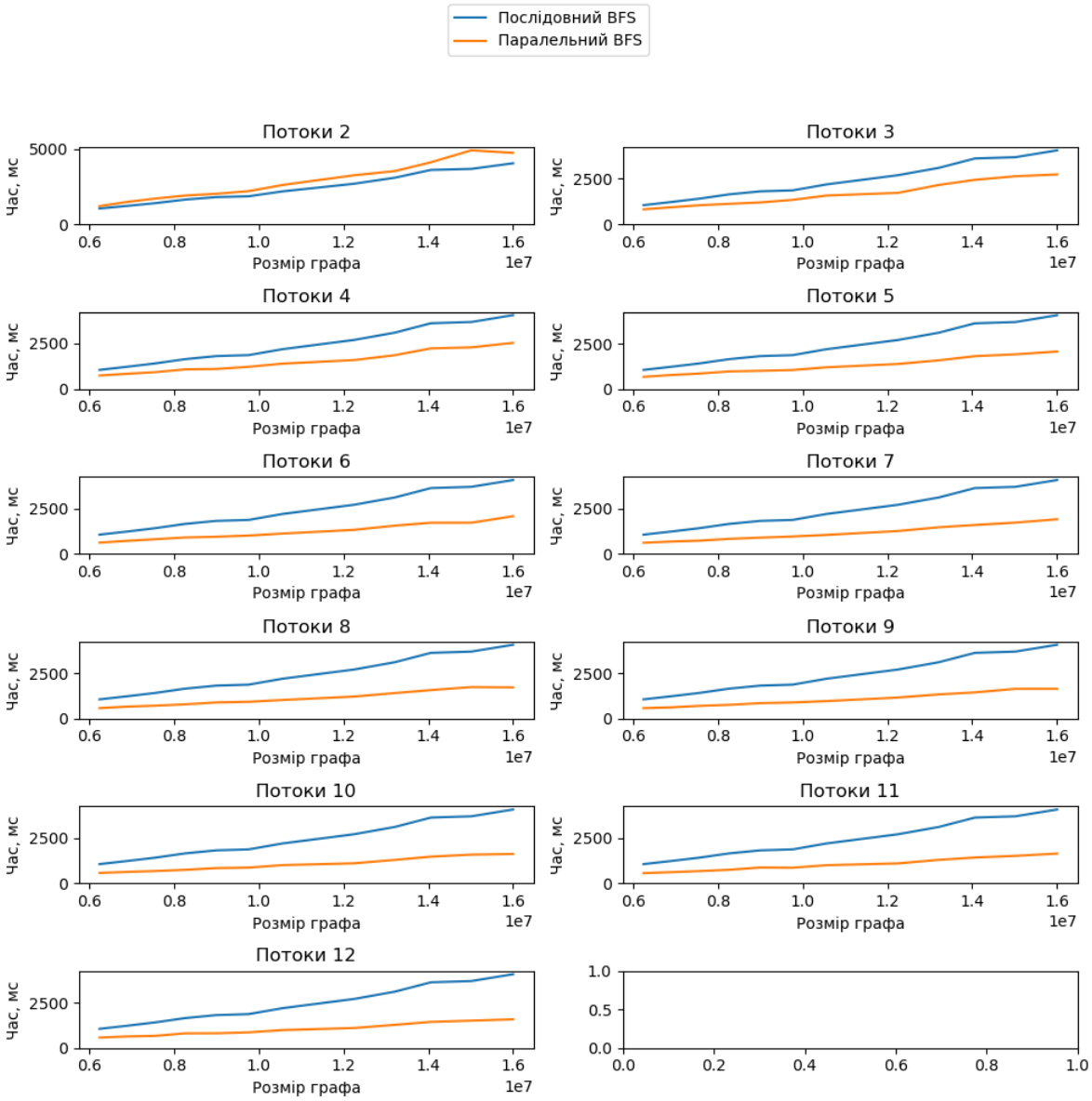

Рисунок 5.1 Залежність часу від розміру графа для різної кількості потоків

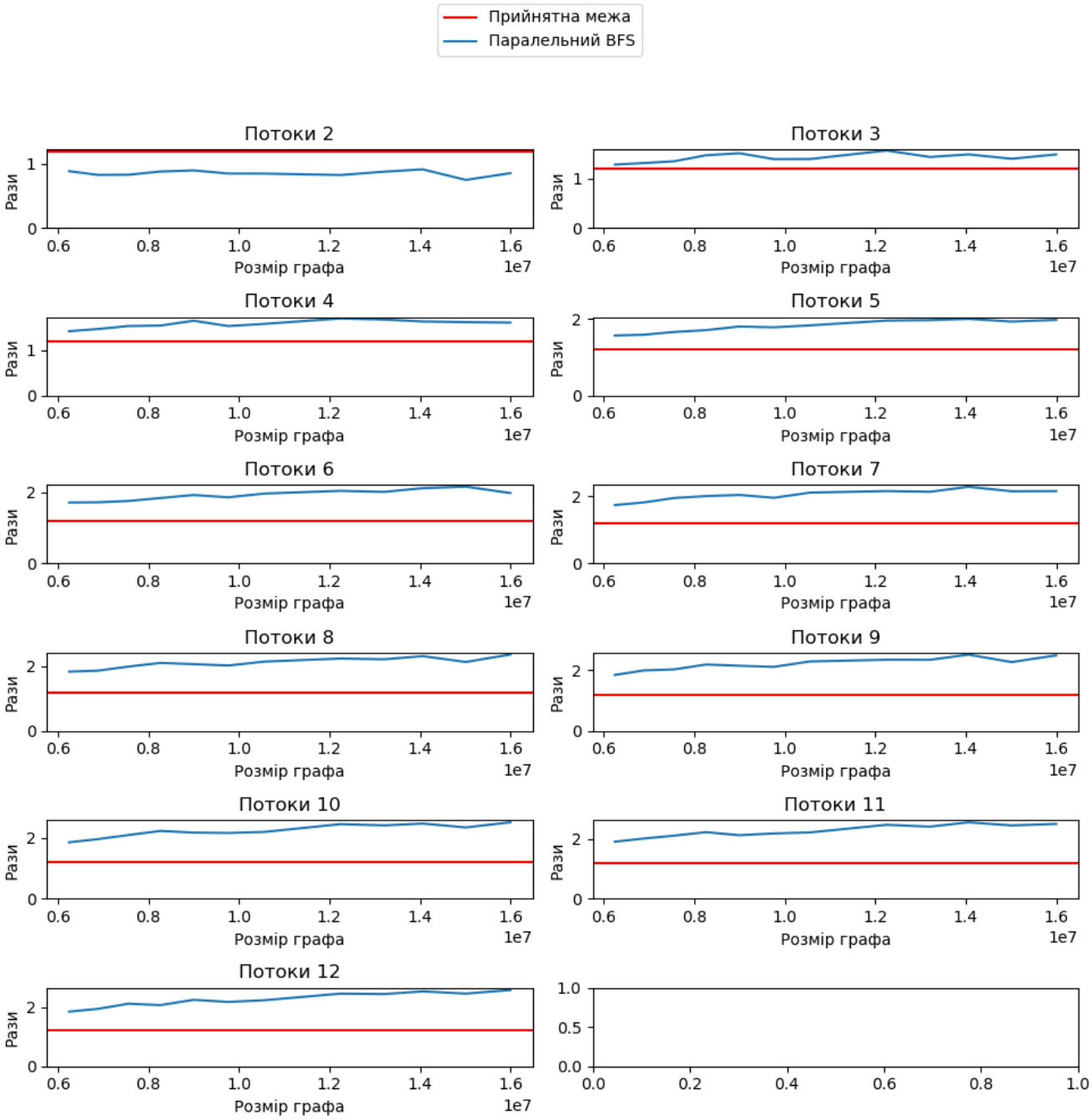

Рисунок 5.2 Залежність прискорення від розміру графа для різної кількості потоків

Паралельний BFS демонструє значне зменшення часу виконання порівняно з послідовним виконанням, особливо для великих графів і при використанні великої кількості потоків. Це підтверджує ефективність паралельної обробки для таких задач.

З зображень видно, що прискорення збільшується з кількістю потоків, але існує межа, після якої додавання додаткових потоків не призводить до значного збільшення прискорення. Це може бути пов'язано з накладними витратами на синхронізацію та управління потоками, які зрештою обмежують загальне прискорення.

ВИСНОВКИ

Виконання курсової роботи дозволило нам глибоко зануритись у вивчення паралельної реалізації алгоритму пошуку в ширину (BFS), що включало теоретичний аналіз алгоритму, детальний опис ключових класів, допоміжних структур даних та їх взаємодії. Процес ретельного тестування виявив помилки, забезпечив виправлення та підтвердження ефективності розробленого рішення.

Аналіз графічних даних, отриманих в ході дослідження, продемонстрував значні переваги застосування паралельного виконання BFS у порівнянні з послідовним. При збільшенні кількості потоків до чотирьох, прискорення склало близько 45%, що свідчить про ефективність паралелізації задачі. При збільшенні кількості потоків до 12 прискорення склало 130%. Оптимальним варіантом виявилося використання восьми потоків, при якому досягнуто прискорення більше ніж у 2 рази порівняно з послідовною реалізацією.

Проте, було також зазначено, що подальше збільшення кількості потоків до дванадцяти призвело лише до незначного покращення продуктивності, що підкреслює існування межі ефективності паралельної обробки для даної задачі. Це обмеження було пов’язано з накладними витратами на синхронізацію та управління потоками, які з часом починають переважувати над вигодами від додавання нових обчислювальних ресурсів.

У підсумку, дослідження підтвердило високу ефективність паралельної реалізації BFS, зокрема, при оптимальній кількості потоків, що дозволяє значно скоротити час обробки великих графів. Результати цієї роботи відкривають нові перспективи для розробки та оптимізації паралельних алгоритмів, сприяючи подальшому прогресу в галузі обчислювальної техніки та програмування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Heineman, G. T., Pollice, G., Selkow, S. Algorithms in a Nutshell. Sebastopol, CA: O'Reilly Media, 2009. Breadth-first search. С. 104.
2. C++ [Електронний ресурс] — https://en.cppreference.com/w/cpp/language.
3. Heineman, G. T., Pollice, G., Selkow, S. Algorithms in a Nutshell. Sebastopol, CA: O'Reilly Media, 2009. Parallel Breadth-first search. С. 107.
4. Std::regular [Електронний ресурс] — https://en.cppreference.com/w/cpp/concepts/regular.
5. std::hash [Електронний ресурс] — https://en.cppreference.com/w/cpp/utility/hash.
6. Jason Turner. C++ Weekly. 2021. Ep. 259 - CRTP: What It Is. Some History and Some Uses. URL: https://www.youtube.com/watch?v=ZQ-8laAr9Dg (дата звернення : 11.01.2024)
7. googletest [Електронний ресурс] — https://github.com/google/googletest.
8. JSON [Електронний ресурс] — https://www.json.org/json-en.html.
9. <thread> [Електронний ресурс] — https://en.cppreference.com/w/cpp/header/thread.
10. <mutex> [Електронний ресурс] — https://en.cppreference.com/w/cpp/header/mutex.
11. <atomic> [Електронний ресурс] — https://en.cppreference.com/w/cpp/header/atomic.
12. Boost.MPI [Електронний ресурс] — https://www.boost.org/doc/libs/1_80_0/doc/html/mpi.html#mpi.introduction.
13. Boost.Asio [Електронний ресурс] — https://www.boost.org/doc/libs/1_84_0/doc/html/boost_asio.html.
14. PostgreSQL [Електронний ресурс] — https://www.postgresql.org/.
15. MS SQL [Електронний ресурс] — https://www.microsoft.com/en/sql-server/sql-server-2019.
16. std::unordered_map [Електронний ресурс] — https://en.cppreference.com/w/cpp/container/unordered_map.
17. std::jthread [Електронний ресурс] — https://en.cppreference.com/w/cpp/thread/jthread.

ДОДАТОК А

ТЕСТУВАННЯ ПАРАЛЕЛЬНОГО АЛГОРИТМУ

Кількість елементів	Мілісекунди	К-ть потоків	Прискорення
6250000	1297	2	0.76946800308404
6250000	785	3	1.27133757961783
6250000	764	4	1.30628272251309
6250000	641	5	1.55694227769111
6250000	635	6	1.57165354330709
6250000	634	7	1.57413249211356
6250000	581	8	1.71772805507745
6250000	583	9	1.71183533447684
6250000	593	10	1.68296795952782
6250000	539	11	1.85157699443414
6250000	540	12	1.84814814814815
6250000	992	2	1.04435483870968
6250000	855	3	1.21169590643275
6250000	760	4	1.36315789473684
6250000	659	5	1.57207890743551
6250000	606	6	1.70957095709571
6250000	581	7	1.78313253012048
6250000	583	8	1.7770154373928
6250000	566	9	1.83038869257951
6250000	560	10	1.85
6250000	602	11	1.72093023255814
6250000	676	12	1.53254437869822
6250000	1353	2	0.766444937176644
6250000	858	3	1.20862470862471
6250000	728	4	1.42445054945055
6250000	657	5	1.57838660578387
6250000	603	6	1.71973466003317
6250000	595	7	1.74285714285714
6250000	571	8	1.81611208406305
6250000	552	9	1.8786231884058
6250000	550	10	1.88545454545455
6250000	553	11	1.875226039783
6250000	551	12	1.88203266787659
6250000	1061	2	1.01790763430726
6250000	823	3	1.31227217496962
6250000	739	4	1.46143437077131
6250000	720	5	1.5
6250000	634	6	1.70347003154574
6250000	582	7	1.85567010309278
6250000	568	8	1.90140845070423
6250000	588	9	1.83673469387755
6250000	588	10	1.83673469387755

6250000	548	11	1.97080291970803
6250000	545	12	1.98165137614679
6250000	1352	2	0.832100591715976
6250000	804	3	1.39925373134328
6250000	744	4	1.51209677419355
6250000	694	5	1.62103746397695
6250000	602	6	1.8687707641196
6250000	648	7	1.73611111111111
6250000	569	8	1.97715289982425
6250000	577	9	1.94974003466205
6250000	556	10	2.02338129496403
6250000	537	11	2.09497206703911
6250000	570	12	1.97368421052632
6890625	1329	2	0.931527464258841
6890625	898	3	1.37861915367483
6890625	829	4	1.49336550060314
6890625	732	5	1.69125683060109
6890625	692	6	1.78901734104046
6890625	679	7	1.82326951399116
6890625	669	8	1.85052316890882
6890625	646	9	1.91640866873065
6890625	657	10	1.88432267884323
6890625	625	11	1.9808
6890625	604	12	2.04966887417219
6890625	1530	2	0.77516339869281
6890625	907	3	1.30760749724366
6890625	790	4	1.50126582278481
6890625	802	5	1.4788029925187
6890625	710	6	1.67042253521127
6890625	661	7	1.79425113464448
6890625	646	8	1.8359133126935
6890625	604	9	1.9635761589404
6890625	629	10	1.88553259141494
6890625	615	11	1.92845528455285
6890625	599	12	1.97996661101836
6890625	1635	2	0.746788990825688
6890625	988	3	1.23582995951417
6890625	840	4	1.45357142857143
6890625	786	5	1.55343511450382
6890625	745	6	1.63892617449664
6890625	674	7	1.81157270029674
6890625	636	8	1.91981132075472
6890625	605	9	2.01818181818182
6890625	605	10	2.01818181818182
6890625	605	11	2.01818181818182
6890625	743	12	1.64333781965007

6890625	1590	2	0.785534591194969
6890625	949	3	1.31612223393045
6890625	900	4	1.38777777777778
6890625	739	5	1.69012178619756
6890625	692	6	1.80491329479769
6890625	648	7	1.92746913580247
6890625	663	8	1.88386123680241
6890625	596	9	2.09563758389262
6890625	621	10	2.01127214170692
6890625	590	11	2.11694915254237
6890625	643	12	1.94245723172628
6890625	1356	2	0.903392330383481
6890625	936	3	1.30876068376068
6890625	833	4	1.47058823529412
6890625	805	5	1.52173913043478
6890625	719	6	1.70375521557719
6890625	707	7	1.73267326732673
6890625	661	8	1.85325264750378
6890625	625	9	1.96
6890625	608	10	2.01480263157895
6890625	613	11	1.99836867862969
6890625	590	12	2.07627118644068
7562500	1679	2	0.853484216795712
7562500	1043	3	1.37392138063279
7562500	903	4	1.58693244739756
7562500	815	5	1.75828220858896
7562500	829	6	1.72858866103739
7562500	715	7	2.0041958041958
7562500	679	8	2.11045655375552
7562500	666	9	2.15165165165165
7562500	643	10	2.22861586314152
7562500	651	11	2.20122887864823
7562500	649	12	2.20801232665639
7562500	1455	2	0.984879725085911
7562500	1105	3	1.29683257918552
7562500	1010	4	1.41881188118812
7562500	901	5	1.59045504994451
7562500	775	6	1.84903225806452
7562500	738	7	1.94173441734417
7562500	705	8	2.03262411347518
7562500	718	9	1.9958217270195
7562500	687	10	2.08588064046579
7562500	689	11	2.07982583454282
7562500	670	12	2.13880597014925
7562500	2044	2	0.719178082191781
7562500	1027	3	1.43135345666991

7562500	845	4	1.7396449704142
7562500	807	5	1.82156133828996
7562500	824	6	1.78398058252427
7562500	730	7	2.01369863013699
7562500	705	8	2.08510638297872
7562500	711	9	2.06751054852321
7562500	671	10	2.19076005961252
7562500	687	11	2.13973799126638
7562500	714	12	2.05882352941176
7562500	1731	2	0.800115540150202
7562500	1063	3	1.30291627469426
7562500	900	4	1.53888888888889
7562500	851	5	1.62749706227967
7562500	831	6	1.66666666666667
7562500	719	7	1.92628650904033
7562500	683	8	2.02781844802343
7562500	671	9	2.06408345752608
7562500	656	10	2.11128048780488
7562500	655	11	2.11450381679389
7562500	659	12	2.10166919575114
7562500	1663	2	0.799158147925436
7562500	1001	3	1.32767232767233
7562500	988	4	1.34514170040486
7562500	887	5	1.49830890642616
7562500	753	6	1.76494023904382
7562500	713	7	1.86395511921459
7562500	773	8	1.71927554980595
7562500	722	9	1.84072022160665
7562500	714	10	1.86134453781513
7562500	668	11	1.98952095808383
7562500	648	12	2.05092592592593
8265625	2552	2	0.71512539184953
8265625	1127	3	1.61934338952973
8265625	979	4	1.86414708886619
8265625	910	5	2.00549450549451
8265625	830	6	2.19879518072289
8265625	796	7	2.2927135678392
8265625	750	8	2.43333333333333
8265625	732	9	2.4931693989071
8265625	725	10	2.51724137931034
8265625	719	11	2.53824756606398
8265625	720	12	2.53472222222222
8265625	1537	2	1.00975927130774
8265625	1285	3	1.20778210116732
8265625	1173	4	1.3231031543052
8265625	972	5	1.59670781893004

8265625	922	6	1.68329718004338
8265625	844	7	1.83886255924171
8265625	816	8	1.90196078431373
8265625	793	9	1.95712484237074
8265625	787	10	1.9720457433291
8265625	735	11	2.11156462585034
8265625	762	12	2.03674540682415
8265625	1917	2	0.828377673448096
8265625	1074	3	1.47858472998138
8265625	988	4	1.60728744939271
8265625	880	5	1.80454545454545
8265625	824	6	1.92718446601942
8265625	783	7	2.02809706257982
8265625	752	8	2.11170212765957
8265625	726	9	2.18732782369146
8265625	713	10	2.22720897615708
8265625	714	11	2.22408963585434
8265625	777	12	2.04375804375804
8265625	1934	2	0.820062047569803
8265625	1063	3	1.49200376293509
8265625	1029	4	1.54130223517979
8265625	991	5	1.60040363269425
8265625	924	6	1.71645021645022
8265625	798	7	1.9874686716792
8265625	762	8	2.08136482939633
8265625	732	9	2.16666666666667
8265625	724	10	2.19060773480663
8265625	729	11	2.1755829903978
8265625	825	12	1.92242424242424
8265625	1629	2	1.03069367710252
8265625	1090	3	1.54036697247706
8265625	1248	4	1.34535256410256
8265625	1100	5	1.52636363636364
8265625	997	6	1.68405215646941
8265625	889	7	1.88863892013498
8265625	835	8	2.01077844311377
8265625	793	9	2.11727616645649
8265625	747	10	2.24765729585007
8265625	816	11	2.05759803921569
8265625	945	12	1.77671957671958
9000000	1966	2	1.10427263479145
9000000	1221	3	1.77805077805078
9000000	1091	4	1.98991750687443
9000000	1127	5	1.92635314995563
9000000	972	6	2.23353909465021
9000000	874	7	2.48398169336384

9000000	835	8	2.6
9000000	841	9	2.58145065398335
9000000	826	10	2.62832929782082
9000000	802	11	2.7069825436409
9000000	797	12	2.72396486825596
9000000	2232	2	0.790322580645161
9000000	1174	3	1.50255536626917
9000000	1097	4	1.60802187784868
9000000	1009	5	1.74826560951437
9000000	931	6	1.89473684210526
9000000	954	7	1.84905660377359
9000000	849	8	2.07773851590106
9000000	828	9	2.1304347826087
9000000	802	10	2.19950124688279
9000000	832	11	2.12019230769231
9000000	790	12	2.23291139240506
9000000	1771	2	0.984754376058724
9000000	1279	3	1.3635652853792
9000000	1106	4	1.57685352622061
9000000	991	5	1.7598385469223
9000000	948	6	1.83966244725738
9000000	944	7	1.84745762711864
9000000	842	8	2.07125890736342
9000000	846	9	2.06146572104019
9000000	940	10	1.85531914893617
9000000	1057	11	1.64995269631031
9000000	873	12	1.99770904925544
9000000	2112	2	0.816287878787879
9000000	1179	3	1.46225614927905
9000000	1116	4	1.54480286738351
9000000	916	5	1.882096069869
9000000	910	6	1.89450549450549
9000000	834	7	2.06714628297362
9000000	1080	8	1.5962962962963
9000000	940	9	1.83404255319149
9000000	819	10	2.1050061050061
9000000	871	11	1.97933409873708
9000000	809	12	2.13102595797281
9000000	2082	2	0.79731027857829
9000000	1159	3	1.43226919758412
9000000	1113	4	1.49146451033243
9000000	985	5	1.68527918781726
9000000	939	6	1.7678381256656
9000000	849	7	1.95524146054181
9000000	826	8	2.00968523002421
9000000	788	9	2.10659898477157

9000000	793	10	2.09331651954603
9000000	777	11	2.13642213642214
9000000	780	12	2.12820512820513
9765625	2369	2	0.814267623469818
9765625	1392	3	1.38577586206897
9765625	1216	4	1.58634868421053
9765625	1026	5	1.88011695906433
9765625	1023	6	1.88563049853372
9765625	960	7	2.009375
9765625	904	8	2.13384955752212
9765625	869	9	2.21979286536249
9765625	846	10	2.28014184397163
9765625	848	11	2.2747641509434
9765625	860	12	2.24302325581395
9765625	2235	2	0.805816554809843
9765625	1349	3	1.33506300963677
9765625	1225	4	1.47020408163265
9765625	1066	5	1.68949343339587
9765625	1002	6	1.79740518962076
9765625	941	7	1.91392136025505
9765625	924	8	1.9491341991342
9765625	892	9	2.01905829596413
9765625	865	10	2.08208092485549
9765625	861	11	2.0917537746806
9765625	855	12	2.10643274853801
9765625	1902	2	0.964773922187171
9765625	1348	3	1.36127596439169
9765625	1220	4	1.50409836065574
9765625	1050	5	1.74761904761905
9765625	990	6	1.85353535353535
9765625	951	7	1.92954784437434
9765625	922	8	1.99023861171367
9765625	884	9	2.07579185520362
9765625	864	10	2.12384259259259
9765625	843	11	2.17674970344009
9765625	860	12	2.13372093023256
9765625	2271	2	0.812417437252312
9765625	1254	3	1.47129186602871
9765625	1240	4	1.48790322580645
9765625	1051	5	1.75547098001903
9765625	1004	6	1.83764940239044
9765625	947	7	1.94825765575502
9765625	923	8	1.99891657638137
9765625	894	9	2.06375838926174
9765625	870	10	2.12068965517241
9765625	873	11	2.11340206185567

9765625	863	12	2.13789107763615
9765625	2258	2	0.851638618246236
9765625	1379	3	1.39448875997099
9765625	1217	4	1.58011503697617
9765625	1054	5	1.82447817836812
9765625	992	6	1.93850806451613
9765625	971	7	1.98043254376931
9765625	932	8	2.06330472103004
9765625	890	9	2.16067415730337
9765625	869	10	2.21288837744534
9765625	859	11	2.23864959254948
9765625	862	12	2.23085846867749
10562500	2299	2	0.92822966507177
10562500	1488	3	1.43413978494624
10562500	1329	4	1.60571858540256
10562500	1183	5	1.80388841927303
10562500	1090	6	1.95779816513761
10562500	1015	7	2.10246305418719
10562500	993	8	2.14904330312185
10562500	948	9	2.25105485232068
10562500	949	10	2.24868282402529
10562500	925	11	2.30702702702703
10562500	929	12	2.29709364908504
10562500	2155	2	0.975406032482599
10562500	1741	3	1.20735209649627
10562500	1518	4	1.38471673254282
10562500	1170	5	1.7965811965812
10562500	1095	6	1.91963470319635
10562500	1010	7	2.08118811881188
10562500	1010	8	2.08118811881188
10562500	944	9	2.22669491525424
10562500	1019	10	2.06280667320903
10562500	956	11	2.19874476987448
10562500	954	12	2.20335429769392
10562500	2963	2	0.762402969962875
10562500	1576	3	1.43337563451777
10562500	1383	4	1.63340563991323
10562500	1206	5	1.87313432835821
10562500	1117	6	2.02238137869293
10562500	1113	7	2.02964959568733
10562500	1062	8	2.1271186440678
10562500	959	9	2.3555787278415
10562500	982	10	2.30040733197556
10562500	1022	11	2.21037181996086
10562500	994	12	2.27263581488934
10562500	2941	2	0.762325739544373

10562500	1620	3	1.38395061728395
10562500	1330	4	1.68571428571429
10562500	1187	5	1.88879528222409
10562500	1169	6	1.91787852865697
10562500	1040	7	2.15576923076923
10562500	1048	8	2.13931297709924
10562500	998	9	2.24649298597194
10562500	1117	10	2.00716204118174
10562500	1145	11	1.95807860262009
10562500	1114	12	2.01256732495512
10562500	2726	2	0.816581071166544
10562500	1483	3	1.50101146325017
10562500	1428	4	1.55882352941176
10562500	1241	5	1.79371474617244
10562500	1102	6	2.01996370235935
10562500	1019	7	2.18449460255152
10562500	991	8	2.24621594349142
10562500	947	9	2.35058078141499
10562500	938	10	2.37313432835821
10562500	932	11	2.38841201716738
10562500	944	12	2.35805084745763
12250000	3149	2	0.822800889171166
12250000	1724	3	1.50290023201856
12250000	1571	4	1.64926798217696
12250000	1387	5	1.86806056236482
12250000	1293	6	2.00386697602475
12250000	1381	7	1.87617668356264
12250000	1378	8	1.88026124818578
12250000	1282	9	2.0210608424337
12250000	1109	10	2.33633904418395
12250000	1089	11	2.37924701561065
12250000	1084	12	2.39022140221402
12250000	3469	2	0.882386855001441
12250000	1694	3	1.80696576151122
12250000	1612	4	1.89888337468983
12250000	1402	5	2.18330955777461
12250000	1442	6	2.12274618585298
12250000	1280	7	2.39140625
12250000	1182	8	2.58967851099831
12250000	1147	9	2.66870095902354
12250000	1110	10	2.75765765765766
12250000	1105	11	2.77013574660633
12250000	1166	12	2.62521440823328
12250000	3303	2	0.800181653042688
12250000	1715	3	1.54110787172012
12250000	1587	4	1.66540642722117

12250000	1382	5	1.91244573082489
12250000	1295	6	2.04092664092664
12250000	1171	7	2.25704526046114
12250000	1167	8	2.26478149100257
12250000	1113	9	2.37466307277628
12250000	1079	10	2.44949026876738
12250000	1094	11	2.41590493601463
12250000	1081	12	2.44495837187789
12250000	3191	2	0.829207145095581
12250000	1797	3	1.47245409015025
12250000	1604	4	1.64962593516209
12250000	1357	5	1.94988946204864
12250000	1283	6	2.06235385814497
12250000	1219	7	2.17063166529943
12250000	1177	8	2.24808836023789
12250000	1138	9	2.32513181019332
12250000	1096	10	2.41423357664234
12250000	1101	11	2.40326975476839
12250000	1081	12	2.4477335800185
12250000	3209	2	0.798067933935806
12250000	1720	3	1.48895348837209
12250000	1600	4	1.600625
12250000	1377	5	1.85984023238925
12250000	1284	6	1.99454828660436
12250000	1225	7	2.09061224489796
12250000	1146	8	2.2347294938918
12250000	1105	9	2.31764705882353
12250000	1104	10	2.31974637681159
12250000	1084	11	2.36254612546125
12250000	1090	12	2.34954128440367
13213225	3662	2	0.804478427089022
13213225	2055	3	1.43357664233577
13213225	1836	4	1.60457516339869
13213225	1473	5	2
13213225	1436	6	2.05153203342618
13213225	1366	7	2.15666178623719
13213225	1279	8	2.30336200156372
13213225	1236	9	2.38349514563107
13213225	1209	10	2.43672456575682
13213225	1188	11	2.47979797979798
13213225	1177	12	2.50297366185217
13213225	3680	2	0.792119565217391
13213225	2129	3	1.36918741193048
13213225	1957	4	1.48952478283086
13213225	1761	5	1.65530948324815
13213225	1460	6	1.99657534246575

13213225	1335	7	2.18352059925094
13213225	1434	8	2.03277545327755
13213225	1357	9	2.14812085482682
13213225	1218	10	2.39326765188834
13213225	1210	11	2.40909090909091
13213225	1196	12	2.43729096989967
13213225	3529	2	0.856333238877869
13213225	2065	3	1.4634382566586
13213225	1705	4	1.77243401759531
13213225	1472	5	2.05298913043478
13213225	1449	6	2.0855762594893
13213225	1342	7	2.25186289120715
13213225	1275	8	2.37019607843137
13213225	1235	9	2.44696356275304
13213225	1232	10	2.45292207792208
13213225	1314	11	2.29984779299848
13213225	1187	12	2.54591406908172
13213225	3033	2	0.982855258819651
13213225	2160	3	1.38009259259259
13213225	1785	4	1.67002801120448
13213225	1576	5	1.89149746192893
13213225	1831	6	1.62807209175314
13213225	1716	7	1.73717948717949
13213225	1570	8	1.89872611464968
13213225	1426	9	2.09046283309958
13213225	1398	10	2.13233190271817
13213225	1366	11	2.18228404099561
13213225	1394	12	2.13845050215208
13213225	3808	2	0.958508403361344
13213225	2407	3	1.51641046946406
13213225	2025	4	1.80246913580247
13213225	1633	5	2.23515003061849
13213225	1576	6	2.31598984771574
13213225	1549	7	2.35635894125242
13213225	1462	8	2.49658002735978
13213225	1389	9	2.62778977681785
13213225	1376	10	2.65261627906977
13213225	1373	11	2.65841223597961
13213225	1424	12	2.56320224719101
14062500	5371	2	0.777508843790728
14062500	2757	3	1.51468988030468
14062500	2134	4	1.95688847235239
14062500	1855	5	2.25121293800539
14062500	1736	6	2.40552995391705
14062500	1570	7	2.65987261146497
14062500	1542	8	2.70817120622568

14062500	1493	9	2.79705291359679
14062500	1453	10	2.87405368203716
14062500	1446	11	2.88796680497925
14062500	1465	12	2.85051194539249
14062500	2922	2	1.23477070499658
14062500	2251	3	1.60284318080853
14062500	2229	4	1.61866307761328
14062500	1827	5	1.97482211275315
14062500	1893	6	1.90596936080296
14062500	1703	7	2.11861421021726
14062500	1733	8	2.08193883439123
14062500	1432	9	2.5195530726257
14062500	1395	10	2.58637992831541
14062500	1404	11	2.56980056980057
14062500	1375	12	2.624
14062500	3336	2	1.04736211031175
14062500	2502	3	1.396482813749
14062500	2535	4	1.37830374753452
14062500	1861	5	1.87748522299839
14062500	1668	6	2.0947242206235
14062500	1594	7	2.19196988707654
14062500	1637	8	2.13439218081857
14062500	1454	9	2.40302613480055
14062500	1524	10	2.29265091863517
14062500	1473	11	2.37202987101154
14062500	1375	12	2.54109090909091
14062500	4616	2	0.736351819757366
14062500	2530	3	1.34347826086957
14062500	2134	4	1.59278350515464
14062500	1852	5	1.835313174946
14062500	1666	6	2.04021608643457
14062500	1504	7	2.25997340425532
14062500	1426	8	2.3835904628331
14062500	1414	9	2.4038189533239
14062500	1565	10	2.17188498402556
14062500	1375	11	2.472
14062500	1605	12	2.11775700934579
14062500	4391	2	0.777954907765885
14062500	2180	3	1.56697247706422
14062500	2155	4	1.58515081206497
14062500	1656	5	2.06280193236715
14062500	1587	6	2.15248897290485
14062500	1558	7	2.19255455712452
14062500	1495	8	2.28494983277592
14062500	1410	9	2.42269503546099
14062500	1394	10	2.45050215208034

14062500	1395	11	2.44874551971326
14062500	1369	12	2.49525200876552
15015625	4665	2	0.792068595927117
15015625	2654	3	1.39223813112283
15015625	2274	4	1.62489006156552
15015625	1933	5	1.91153647180548
15015625	1797	6	2.05620478575403
15015625	1824	7	2.02576754385965
15015625	1877	8	1.96856686201385
15015625	1778	9	2.07817772778403
15015625	1721	10	2.14700755374782
15015625	1463	11	2.52563226247437
15015625	1484	12	2.48989218328841
15015625	5002	2	0.735705717712915
15015625	2543	3	1.44710971293748
15015625	2173	4	1.69351127473539
15015625	1918	5	1.91866527632951
15015625	1761	6	2.08972174900625
15015625	1682	7	2.18787158145065
15015625	1605	8	2.29283489096573
15015625	1526	9	2.41153342070773
15015625	1523	10	2.41628365068943
15015625	1570	11	2.34394904458599
15015625	1593	12	2.31010671688638
15015625	5103	2	0.722712130119538
15015625	2729	3	1.35141077317699
15015625	2431	4	1.51707116412999
15015625	1882	5	1.9596174282678
15015625	1576	6	2.34010152284264
15015625	1646	7	2.24058323207776
15015625	1715	8	2.15043731778426
15015625	1601	9	2.30356027482823
15015625	1490	10	2.4751677852349
15015625	1494	11	2.46854082998661
15015625	1444	12	2.55401662049861
16000000	4528	2	0.882067137809187
16000000	2714	3	1.47162859248342
16000000	2490	4	1.60401606425703
16000000	1937	5	2.06195147134744
16000000	1969	6	2.02844083291011
16000000	1813	7	2.20297848869277
16000000	1706	8	2.3411488862837
16000000	1644	9	2.4294403892944
16000000	1606	10	2.48692403486924
16000000	1571	11	2.54232972628899
16000000	1557	12	2.56518946692357

16000000	4718	2	0.835311572700297
16000000	2795	3	1.41001788908766
16000000	2500	4	1.5764
16000000	2217	5	1.77762742444745
16000000	1985	6	1.98539042821159
16000000	1798	7	2.19187986651835
16000000	1700	8	2.31823529411765
16000000	1621	9	2.43121529919803
16000000	1651	10	2.3870381586917
16000000	1863	11	2.1154052603328
16000000	1612	12	2.44478908188586
16000000	5062	2	0.883840379296721
16000000	2832	3	1.57980225988701
16000000	2828	4	1.58203677510608
16000000	2203	5	2.03086699954607
16000000	1952	6	2.29200819672131
16000000	1809	7	2.47318960751797
16000000	1701	8	2.63021751910641
16000000	1621	9	2.76002467612585
16000000	1601	10	2.7945034353529
16000000	1572	11	2.84605597964377
16000000	1569	12	2.8514977692798
16000000	4456	2	0.879488330341113
16000000	2833	3	1.3833392163784
16000000	2492	4	1.57263242375602
16000000	2086	5	1.87871524448706
16000000	1948	6	2.01180698151951
16000000	1802	7	2.17480577136515
16000000	1736	8	2.25748847926267
16000000	1646	9	2.38092345078979
16000000	1592	10	2.46168341708543
16000000	1586	11	2.47099621689786
16000000	1573	12	2.49141767323585
16000000	5008	2	0.796126198083067
16000000	2529	3	1.57651245551601
16000000	2386	4	1.6709974853311
16000000	1886	5	2.11399787910923
16000000	2498	6	1.59607686148919
16000000	2291	7	1.7402880838062
16000000	1739	8	2.29269695227142
16000000	1646	9	2.42223572296476
16000000	1628	10	2.4490171990172
16000000	1588	11	2.51070528967254
16000000	1585	12	2.51545741324921

ДОДАТОК Б

КОД ПРОГРАМИ

```
// ../Include/ParallelBFS/TBaseBFSMixin.hpp

#ifndef PARALLELBFS_TBASEBFSMIXIN_HPP
#define PARALLELBFS_TBASEBFSMIXIN_HPP

#include <concepts>
#include <unordered_map>
#include <vector>
#include <algorithm>

namespace bfs {

template<typename T>
concept CBFSUsable = std::regular<T> and requires(T value) {
      {std::hash<T>{}(value)} -> std::same_as<std::size_t>;
};

template<CBFSUsable T>
using AGraph = std::unordered_map<T, std::vector<T>>;

template<CBFSUsable T, typename Derived>
class TBaseBFSMixin {
public:
template<typename... Args>
static std::optional<std::vector<T>> Do(const AGraph<T>& graph, const T& start,
const T& end, Args&&... args);

protected:
TBaseBFSMixin(const AGraph<T>& graph, const T& start, const T& end);

protected:
std::optional<std::vector<T>> Execute();

protected:
const Derived* self() const;
Derived* self();

protected:
template<typename ValueType>
std::vector<T>    DeterminePath(const    std::unordered_map<T,    ValueType>&
predecessorNodes) const;
```

```
protected:
const AGraph<T>& m_refGraph;
const T& m_refStart;
const T& m_refEnd;
};

template<CBFSUsable T, typename Derived>
template<typename... Args>
std::optional<std::vector<T>> TBaseBFSMixin<T, Derived>::Do(const AGraph<T>&
graph, const T& start, const T& end, Args&&... args) {
auto alg = Derived(graph, start, end, std::forward<Args>(args)...);
return alg.Execute();
}

template<CBFSUsable T, typename Derived>
TBaseBFSMixin<T, Derived>::TBaseBFSMixin(const AGraph<T>& graph, const T&
start, const T& end)
: m_refGraph{graph}, m_refStart{start}, m_refEnd{end} {}

template<CBFSUsable T, typename Derived>
std::optional<std::vector<T>> TBaseBFSMixin<T, Derived>::Execute() {
if(m_refStart == m_refEnd) return std::vector{m_refStart, m_refEnd};
const auto result = self()->PredecessorNodesImpl();
if(not result) return std::nullopt;
return DeterminePath(result.value());
}

template<CBFSUsable T, typename Derived>
const Derived* TBaseBFSMixin<T, Derived>::self() const {
return static_cast<const Derived*>(this);
}

template<CBFSUsable T, typename Derived>
Derived* TBaseBFSMixin<T, Derived>::self() {
return static_cast<Derived*>(this);
}

template<CBFSUsable T, typename Derived>
template<typename ValueType>
std::vector<T> TBaseBFSMixin<T, Derived>::DeterminePath(
const std::unordered_map<T, ValueType>& predecessorNodes) const {
auto path = std::vector<T>{this->m_refEnd};
auto currentNode = path.front();
```

```
while(currentNode != this->m_refStart) {
      currentNode = predecessorNodes.at(currentNode).second;
      path.push_back(currentNode);
}
std::reverse(path.begin(), path.end());
return path;
}

}

#endif //PARALLELBFS_TBASEBFSMIXIN_HPP


// ../Include/ParallelBFS/TPipes.hpp

#ifndef PARALLELBFS_TPIPES_HPP
#define PARALLELBFS_TPIPES_HPP

#include <concepts>
#include <memory>

namespace bfs {

template<typename T>
concept CPipeUsable = std::default_initializable<T> and std::movable<T>;

template<CPipeUsable T>
class TPipeReader;

template<CPipeUsable T>
class TPipeWriter;

template<CPipeUsable T>
class TPipeChannel;

template<CPipeUsable T>
class TPipeReader {
friend class TPipeChannel<T>;

public:
~TPipeReader()=default;
TPipeReader(TPipeReader&& other) noexcept;
TPipeReader& operator=(TPipeReader&& other) noexcept;
```

```
	public:
	T Read() const;

	protected:
	TPipeReader() = default;
	TPipeReader(const std::shared_ptr<std::pair<T, std::atomic_flag>>& data);
	TPipeReader(const TPipeReader&) = delete;
	TPipeReader& operator=(const TPipeReader&) = delete;

	protected:
	std::shared_ptr<std::pair<T, std::atomic_flag>> m_pData = nullptr;
	};

	template<CPipeUsable T>
	TPipeReader<T>::TPipeReader(const                  std::shared_ptr<std::pair<T,
std::atomic_flag>>& data)
	: m_pData{data} {
	}

	template<CPipeUsable T>
	TPipeReader<T>::TPipeReader(TPipeReader&& other) noexcept
	: m_pData{std::move(other.m_pData)} {
	}

	template<CPipeUsable T>
	TPipeReader<T>& TPipeReader<T>::operator=(TPipeReader&& other) noexcept {
	this->m_pData = std::move(other.m_pData);
	return *this;
	}

	template<CPipeUsable T>
	T TPipeReader<T>::Read() const {
	m_pData->second.wait(false);
	auto inner = std::move(m_pData->first);
	m_pData->second.clear();
	m_pData->second.notify_one();
	return inner;
	}

	template<CPipeUsable T>
	class TPipeWriter {
	friend class TPipeChannel<T>;

	public:
```

```
	~TPipeWriter()=default;
	TPipeWriter(TPipeWriter&& other) noexcept;
	TPipeWriter& operator=(TPipeWriter&& other) noexcept;

	public:
	void Write(T&& value) const;

	protected:
	TPipeWriter() = default;
	TPipeWriter(const std::shared_ptr<std::pair<T, std::atomic_flag>>& data);
	TPipeWriter(const TPipeWriter&) = delete;
	TPipeWriter operator=(const TPipeWriter&) = delete;

	protected:
	std::shared_ptr<std::pair<T, std::atomic_flag>> m_pData = nullptr;
	};

	template<CPipeUsable T>
	TPipeWriter<T>::TPipeWriter(const                 std::shared_ptr<std::pair<T,
std::atomic_flag>>& data)
	: m_pData{data} {
	}

	template<CPipeUsable T>
	TPipeWriter<T>::TPipeWriter(TPipeWriter&& other) noexcept
	: m_pData{std::move(other.m_pData)} {
	}

	template<CPipeUsable T>
	TPipeWriter<T>& TPipeWriter<T>::operator=(TPipeWriter&& other) noexcept {
	this->m_pData = std::move(other.m_pData);
	return *this;
	}

	template<CPipeUsable T>
	void TPipeWriter<T>::Write(T&& value) const {
	m_pData->second.wait(true);
	m_pData->first = std::move(value);
	m_pData->second.test_and_set();
	m_pData->second.notify_one();
	}

	template<CPipeUsable T>
	class TPipeChannel {
```

```
public:
TPipeChannel();

public:
TPipeWriter<T> Writer;
TPipeReader<T> Reader;
};

template<CPipeUsable T>
TPipeChannel<T>::TPipeChannel() {
auto data = std::make_shared<std::pair<T, std::atomic_flag>>();
Writer = TPipeWriter(data);
Reader = TPipeReader(std::move(data));
}

}

#endif //PARALLELBFS_TPIPES_HPP

// ../Include/ParallelBFS/TSequentialBFS.hpp

#ifndef PARALLELBFS_TSEQUENTIALBFS_HPP
#define PARALLELBFS_TSEQUENTIALBFS_HPP

#include <queue>
#include <unordered_set>
#include <ParallelBFS/TBaseBFSMixin.hpp>

namespace bfs {

template<CBFSUsable T>
class TSequentialBFS : public TBaseBFSMixin<T, TSequentialBFS<T>> {
friend class TBaseBFSMixin<T, TSequentialBFS<T>>;

protected:
TSequentialBFS(const AGraph<T>& graph, const T& start, const T& end);

protected:
using AVisitorMap = std::unordered_map<T, std::pair<bool, T>>;
std::optional<AVisitorMap> PredecessorNodesImpl() const;

protected:
AVisitorMap CreateVisitorMap() const;
};
```

```
	template<CBFSUsable T>
	TSequentialBFS<T>::TSequentialBFS(const AGraph<T>& graph, const T& start, const
T& end)
	: TBaseBFSMixin<T, TSequentialBFS>(graph, start, end) {}

	template<CBFSUsable T>
	std::optional<typename TSequentialBFS<T>::AVisitorMap>
	    TSequentialBFS<T>::PredecessorNodesImpl() const {
	auto queue = std::queue<T>({this->m_refStart});
	auto visitorMap = CreateVisitorMap();
	auto isFoundEndNode = false;
	while(not queue.empty() and not isFoundEndNode) {
		const auto currentNode = std::move(queue.front());
		queue.pop();
		for(const auto& neighbour : this->m_refGraph.at(currentNode)) {
			const auto neighbourIt = visitorMap.find(neighbour);
			if(not neighbourIt->second.first) {
				neighbourIt->second.first = true;
				neighbourIt->second.second = currentNode;
				if(neighbour == this->m_refEnd) {
					isFoundEndNode = true;
					break;
				}
				queue.push(neighbour);
			}
		}
	}
	if(not isFoundEndNode) return std::nullopt;
	return visitorMap;
	}

	template<CBFSUsable T>
	TSequentialBFS<T>::AVisitorMap TSequentialBFS<T>::CreateVisitorMap() const {
	auto visitorMap = std::unordered_map<T, std::pair<bool, T>>();
	visitorMap.reserve(this->m_refGraph.size());
	for(const auto& [key, _] : this->m_refGraph) {
		visitorMap.insert_or_assign(key, std::make_pair(false, T()));
	}
	return visitorMap;
	}

	}
```

```
#endif //PARALLELBFS_TSEQUENTIALBFS_HPP


// ../Include/ParallelBFS/THelpers.hpp

#ifndef PARALLELBFS_THELPERS_HPP
#define PARALLELBFS_THELPERS_HPP

#include <variant>
#include <string_view>
#include <source_location>

namespace bfs {

template<typename VariantType, typename T, std::size_t index = 0>
constexpr std::size_t VariantIndex() {
static_assert(std::variant_size_v<VariantType> > index, "Type not found in variant");
if constexpr (index == std::variant_size_v<VariantType>) {
	return index;
} else if constexpr (std::is_same_v<std::variant_alternative_t<index, VariantType>, T>) {
	return index;
} else {
	return VariantIndex<VariantType, T, index + 1>();
}
}

namespace lr {
enum class NLevel {
	Info,
	Warn,
	Error
};

void Log(
	const std::string_view message,
	const NLevel level,
	std::source_location location = std::source_location::current()
);

void LogInfo(
	const std::string_view message,
	std::source_location location = std::source_location::current()
```

```
);

void LogWarn(
      const std::string_view message,
      std::source_location location = std::source_location::current()
);

void LogError(
      const std::string_view message,
      std::source_location location = std::source_location::current()
);

[[noreturn]] void Error(
      const std::string_view message,
      std::source_location location = std::source_location::current()
);

[[noreturn]] void UnsopportedCaseError(
      std::source_location location = std::source_location::current()
);

}

}

#endif //PARALLELBFS_THELPERS_HPP


// ../Include/ParallelBFS/TCommunicationBFS.hpp

#ifndef PARALLELBFS_TCOMMUNICATIONBFS_HPP
#define PARALLELBFS_TCOMMUNICATIONBFS_HPP

#include <ParallelBFS/THelpers.hpp>
#include <ParallelBFS/TPipes.hpp>
#include <ParallelBFS/TDeque.hpp>
#include <ParallelBFS/TBaseBFSMixin.hpp>

#include <thread>

namespace bfs {

template<CBFSUsable T>
class TCommunicationBFS : public TBaseBFSMixin<T, TCommunicationBFS<T>> {
```

```
	protected:
	friend class TBaseBFSMixin<T, TCommunicationBFS<T>>;

	protected:
	TCommunicationBFS(const AGraph<T>& graph, const T& start, const T& end, const unsigned threadsNum);

	protected:
	using AVisitorMap = std::unordered_map<T, std::pair<std::atomic_flag, T>>;

	protected:
	AVisitorMap CreateVisitorMap() const;

	protected:
	const unsigned m_uThreadsNum = 0;

	protected:
	std::optional<AVisitorMap> PredecessorNodesImpl() const;

	protected:
	// Messages
	struct SContinueIteration {};
	struct SEndNodeFound {};
	struct SAllNodesEnqueued {};
	struct SQueueView {
		const TDeque<T>* Deque;
		size_t Begin;
		size_t End;
	};
	struct SFrontier {
		std::vector<T> Data;
	};

	using AParentMessage = std::variant<
		SEndNodeFound,
		SAllNodesEnqueued,
		SQueueView>;

	using AChildrenMessage = std::variant<
		SEndNodeFound,
		SFrontier>;

	using ACommunicationResult = std::variant<
		SAllNodesEnqueued,
```

```
            SEndNodeFound
        >;

        using AIterationResult = std::variant<
            SEndNodeFound,
            SContinueIteration
        >;

        protected:
        ACommunicationResult Communicate(
            TDeque<T>& deque,
            size_t& totalEnqueuedNum,
            AVisitorMap& visitorMap,
            std::vector<TPipeWriter<AParentMessage>>& senders,
            std::vector<TPipeReader<AChildrenMessage>>& listeners
        ) const;

        AChildrenMessage DoPartialWork(
            const SQueueView& queueView,
            AVisitorMap& visitorMap) const;

        void ChildThreadWork(
            const TPipeWriter<AChildrenMessage>& childSender,
            const TPipeReader<AParentMessage>& parentListener,
            AVisitorMap& visitorMap
        ) const;

        AChildrenMessage IterateWork(
            const TDeque<T>& deque,
            const std::vector<TPipeWriter<AParentMessage>>& senders,
            AVisitorMap& visitorMap
        ) const;

        auto ProcessIterationResult(
            TDeque<T>& deque,
            AChildrenMessage&& partialResult,
            const std::vector<TPipeWriter<AParentMessage>>& senders,
            size_t& totalEnqueued
        ) const -> AIterationResult;

        template<typename MessageType>
        MessageType  SendMessageToAll(const  std::vector<TPipeWriter<AParentMessage>>&
senders) const;
        };
```

```
template<CBFSUsable T>
TCommunicationBFS<T>::TCommunicationBFS(
const AGraph<T>& graph,
const T& start,
const T& end,
const unsigned threadsNum
) :   TBaseBFSMixin<T, TCommunicationBFS>(graph, start, end),
m_uThreadsNum{threadsNum} {}

template<CBFSUsable T>
std::unordered_map<T, std::pair<std::atomic_flag, T>>
TCommunicationBFS<T>::CreateVisitorMap() const {
auto visitorMap = std::unordered_map<T, std::pair<std::atomic_flag, T>>();
visitorMap.reserve(this->m_refGraph.size());
for(const auto& [key, _] : this->m_refGraph) {
      auto [it, isEnqueued] = visitorMap.emplace(std::piecewise_construct,
            std::forward_as_tuple(key), std::forward_as_tuple());
      it->second.first.clear();
}
return visitorMap;
}

template<CBFSUsable T>
std::optional<typename TCommunicationBFS<T>::AVisitorMap>
    TCommunicationBFS<T>::PredecessorNodesImpl() const {

auto deque = TDeque<T>();
deque.Push({this->m_refStart});
size_t totalEnqueuedNum = 0;
auto visitorMap = this->CreateVisitorMap();
auto senders = std::vector<TPipeWriter<AParentMessage>>();
auto listeners = std::vector<TPipeReader<AChildrenMessage>>();

const auto result = Communicate(deque, totalEnqueuedNum,
      visitorMap, senders, listeners);

switch(result.index()) {
      case VariantIndex<ACommunicationResult, SAllNodesEnqueued>(): {
            return std::nullopt;
      }
      case VariantIndex<ACommunicationResult, SEndNodeFound>(): {
            return visitorMap;
      }
```

```
            default: {
                  lr::UnsopportedCaseError();
            }
      }
      }

      template<CBFSUsable T>
      auto TCommunicationBFS<T>::Communicate(
            TDeque<T>& deque,
            size_t& totalEnqueuedNum,
            AVisitorMap& visitorMap,
            std::vector<TPipeWriter<AParentMessage>>& senders,
            std::vector<TPipeReader<AChildrenMessage>>& listeners
      ) const -> ACommunicationResult {

      auto threads = std::vector<std::jthread>();
      for(auto i = 0u; i < this->m_uThreadsNum - 1; ++i) {
            auto [parentSender, parentListener] = TPipeChannel<AParentMessage>();
            auto            [childrenSender,            childrenListener]           =
TPipeChannel<AChildrenMessage>();
            senders.push_back(std::move(parentSender));
            listeners.push_back(std::move(childrenListener));
            threads.emplace_back([this,
                  sender=std::move(childrenSender),
                  listener=std::move(parentListener), &visitorMap] {
                  ChildThreadWork(sender, listener, visitorMap);
            });
      }

      while(true) {
            auto newDeque = TDeque<T>();
            {
                  auto partRes = IterateWork(deque, senders, visitorMap);
                  const auto iterResult = ProcessIterationResult(newDeque,
                        std::move(partRes), senders, totalEnqueuedNum);
                  if(std::holds_alternative<SEndNodeFound>(iterResult)) {
                        return SEndNodeFound{};
                  }
            }
            for(auto& l : listeners) {
                  auto partRes = l.Read();
                  auto iterResult = ProcessIterationResult(newDeque,
                        std::move(partRes), senders, totalEnqueuedNum);
                  if(std::holds_alternative<SEndNodeFound>(iterResult)) {
```

```
                  return SEndNodeFound{};
            }
      }
      deque = std::move(newDeque);
      if(totalEnqueuedNum >= this->m_refGraph.size()) {
            return SendMessageToAll<SAllNodesEnqueued>(senders);
      }
}

return SAllNodesEnqueued{};
}

template<CBFSUsable T>
auto TCommunicationBFS<T>::ProcessIterationResult(
TDeque<T>& deque,
AChildrenMessage&& partialResult,
const std::vector<TPipeWriter<AParentMessage>>& senders,
size_t& totalEnqueued
) const -> AIterationResult {
switch(partialResult.index()) {
      case VariantIndex<AChildrenMessage, SEndNodeFound>(): {
            return SendMessageToAll<SEndNodeFound>(senders);
      }
      case VariantIndex<AChildrenMessage, SFrontier>(): {
            auto& frontier = std::get<SFrontier>(partialResult);
            totalEnqueued += frontier.Data.size();
            if(not frontier.Data.empty()) {
                  deque.Push(std::move(frontier.Data));
            }
            return SContinueIteration{};
      }
      default: {
            lr::UnsopportedCaseError();
      }
}
}

template<CBFSUsable T>
auto TCommunicationBFS<T>::DoPartialWork(
const SQueueView& queueView,
AVisitorMap& visitorMap
) const -> AChildrenMessage {

auto frontier = SFrontier();
```

```
auto isEndNodeFound = false;
queueView.Deque->Loop(queueView.Begin, queueView.End,
      [this, &frontier, &isEndNodeFound, &visitorMap](const T& node) {
      for(const auto& neighbour : this->m_refGraph.at(node)) {
            const auto neighbourIt = visitorMap.find(neighbour);
            if(neighbourIt->second.first.test_and_set())
                  continue;
            neighbourIt->second.second = node;
            if(neighbour == this->m_refEnd) {
                  isEndNodeFound = true;
                  return;
            }
            frontier.Data.push_back(neighbour);
      }
});

if(isEndNodeFound) return SEndNodeFound{};

return frontier;
}

template<CBFSUsable T>
void TCommunicationBFS<T>::ChildThreadWork(
const TPipeWriter<AChildrenMessage>& childSender,
const TPipeReader<AParentMessage>& parentListener,
AVisitorMap& visitorMap
) const {
while(true) {
      const auto parentMessage = parentListener.Read();
      switch(parentMessage.index()) {
            case VariantIndex<AParentMessage, SEndNodeFound>():
            case VariantIndex<AParentMessage, SAllNodesEnqueued>(): {
                  return;
            }
            case VariantIndex<AParentMessage, SQueueView>(): {
                  const auto& queueView = std::get<SQueueView>(parentMessage);
                  childSender.Write(DoPartialWork(queueView, visitorMap));
                  break;
            }
            default: {
                  lr::UnsopportedCaseError();
            }
      }
}
```

```
}

template<CBFSUsable T>
auto TCommunicationBFS<T>::IterateWork(
const TDeque<T>& deque,
const std::vector<TPipeWriter<AParentMessage>>& senders,
AVisitorMap& visitorMap
) const -> AChildrenMessage {
const auto dequeSize = deque.Size();
const auto step = dequeSize / this->m_uThreadsNum;
const auto remainder = dequeSize % this->m_uThreadsNum;
for(size_t t = 0, index = 0; t < this->m_uThreadsNum; ++t) {
      auto queueView = SQueueView{};
      queueView.Deque = &deque;
      if(index >= dequeSize) {
            queueView.Begin = dequeSize;
            queueView.End = dequeSize;
      } else {
            const auto localStep = t < remainder ? step + 1 : step;
            queueView.Begin = index;
            queueView.End = index + localStep;
            index += localStep;
      }
      if(t == this->m_uThreadsNum - 1) {
            return DoPartialWork(queueView, visitorMap);
      } else {
            senders[t].Write(queueView);
      }
}
return SEndNodeFound{};
}

template<CBFSUsable T>
template<typename MessageType>
MessageType TCommunicationBFS<T>::SendMessageToAll(
const std::vector<TPipeWriter<AParentMessage>>& senders
) const {
auto message = MessageType{};
for(auto& s : senders) {
      s.Write(message);
}
return MessageType{};
}
```

```
}

#endif //PARALLELBFS_TCOMMUNICATIONBFS_HPP

// ../Include/ParallelBFS/TDeque.hpp

#ifndef PARALLELBFS_TDEQUE_HPP
#define PARALLELBFS_TDEQUE_HPP

#include <vector>
#include <optional>

namespace bfs {

template<typename T>
class TDeque {
public:
TDeque()=default;

public:
void Push(std::vector<T>&& value);
std::size_t Size() const noexcept;
void Loop(const size_t begin, const size_t end,
      const std::function<void(const T& el)>& func) const;

protected:
std::vector<std::vector<T>> m_vData;
};

template<typename T>
void TDeque<T>::Push(std::vector<T>&& value) {
m_vData.push_back(std::move(value));
}

template<typename T>
std::size_t TDeque<T>::Size() const noexcept {
auto size = size_t(0);
for(const auto& el : m_vData) {
      size += el.size();
}
return size;
}

template<typename T>
```

```
void TDeque<T>::Loop(const size_t begin, const size_t end,
const std::function<void(const T&)>& func) const {
if(begin == end) return;
auto vectorIt = m_vData.begin();
auto elIt = m_vData.begin()->begin();

auto delay = begin;
auto dist = vectorIt->end() - elIt;
while(delay >= dist) {
      delay -= dist;
      ++vectorIt;
      elIt = vectorIt->begin();
      dist = vectorIt->end() - elIt;
}

elIt += delay;

for(auto i = begin; i < end; ++i, ++elIt) {
      if(elIt == vectorIt->end()) {
            ++vectorIt;
            elIt = vectorIt->begin();
      }
      func(*elIt);
}
}

}

#endif //PARALLELBFS_TDEQUE_HPP


// ../Tests/Deque.cpp

#include <gtest/gtest.h>
#include <ParallelBFS/TDeque.hpp>

class TDequeTest
: public testing::TestWithParam<std::array<int, 2>> {
protected:
static void SetUpTestSuite();
static bfs::TDeque<int> s_vDeque;
};

void TDequeTest::SetUpTestSuite() {
```

```
s_vDeque.Push({0, 1, 2, 3, 4});
s_vDeque.Push({5, 6, 7, 8, 9});
s_vDeque.Push({10, 11, 12, 13, 14});
}

bfs::TDeque<int> TDequeTest::s_vDeque = bfs::TDeque<int>();

INSTANTIATE_TEST_SUITE_P(Loop, TDequeTest,
testing::Values(std::array{0, 15}, std::array{3, 12}));

TEST_P(TDequeTest, Loop) {
auto [beginIt, endIt] = GetParam();
s_vDeque.Loop(beginIt, endIt, [&beginIt](const auto& el) {
      EXPECT_EQ(el, beginIt);
      ++beginIt;
});
EXPECT_EQ(beginIt, endIt);
}

// ../Tests/Pipes.cpp

#include <thread>
#include <gtest/gtest.h>
#include <ParallelBFS/TPipes.hpp>

TEST(Pipes, Transfer) {
auto [w, r] = bfs::TPipeChannel<int>();

auto sender = std::jthread([ww=std::move(w)]() {
      using namespace std::chrono_literals;
      std::this_thread::sleep_for(4s);
      ww.Write(10);
});

auto listener = std::jthread([rr=std::move(r)]() {
      EXPECT_EQ(rr.Read(), 10);
});
}

// ../Tests/Benchmark.cpp

#include <format>
#include <ranges>
#include <fstream>
```

```
#include <filesystem>
#include <gtest/gtest.h>
#include <ParallelBFS/TSequentialBFS.hpp>
#include <ParallelBFS/TCommunicationBFS.hpp>

class TTestBFSFixture : public ::testing::Test {
protected:
static std::unordered_map<unsigned, std::vector<unsigned>> Create2DGrid(const unsigned int size);
static bool IsPathValid(const std::vector<unsigned int>& path, const bfs::AGraph<unsigned int>& graph);
static unsigned GetLastIndex(const unsigned size);
static void WriteToReport(const std::string& str);
};

std::unordered_map<unsigned, std::vector<unsigned>>
TTestBFSFixture::Create2DGrid(const unsigned int size) {
auto grid = std::unordered_map<unsigned, std::vector<unsigned>>();
const auto totalSize = size * size;
grid.reserve(totalSize);
for(auto index = 0u; index < totalSize; ++index) {
      const auto x = static_cast<int>(index % size);
      const auto y = static_cast<int>(index / size);
      const auto utmost = static_cast<int>(size) - 1;
      auto neighbourIndexes = std::vector<unsigned>();
      for(auto deltaY = -1; deltaY <= 1; ++deltaY) {
            const auto newY = y + deltaY;
            if(newY < 0 or newY > utmost) continue;
            const auto base = static_cast<unsigned>(newY) * size;
            for(auto deltaX = -1; deltaX <= 1; ++deltaX) {
                  if(deltaY == 0 && deltaX == 0) continue;
                  const auto newX = x + deltaX;
                  if(newX < 0 or newX > utmost) continue;
                  const auto offset = static_cast<unsigned>(newX);
                  neighbourIndexes.push_back(base + offset);
            }
      }
      grid.insert_or_assign(index, neighbourIndexes);
}
return grid;
}

bool TTestBFSFixture::IsPathValid(
const std::vector<unsigned int>& path,
```

```
	const bfs::AGraph<unsigned int>& graph) {
	for(const auto& [start, end] : path | std::views::pairwise) {
		const auto it = graph.find(start);
		if(it == graph.end()) {
			return false;
		}
		const auto isContain = std::ranges::contains(it->second, end);
		if(not isContain) {
			return false;
		}
	}
	return true;
	}

	unsigned TTestBFSFixture::GetLastIndex(const unsigned size) {
	return (size - 1) * size + size - 1;
	}

	void TTestBFSFixture::WriteToReport(const std::string& str) {
	std::ofstream("Benchmark.txt", std::ios::app) << str << std::endl;
	}

	TEST_F(TTestBFSFixture, Test) {
	constexpr auto totalRepeats = 1u;
	const auto sizes = std::vector<unsigned>{2500, 2625, 2750, 2875, 3000, 3125,
3250, 3500, 3635, 3750, 3875, 4000};
	// const auto sizes = std::vector<unsigned>{1000, 1200, 1300, 1400};
	const auto threadsNums = std::vector<unsigned>{2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12};
	for(const auto size : sizes) {
		const auto grid = Create2DGrid(size);
		const auto lastIndex = GetLastIndex(size);
		for(auto i = 0u; i < totalRepeats; ++i) {
			const auto [sequentialMillis, singleRes] = [&grid, &lastIndex,
&size]() {
				const auto start = std::chrono::system_clock::now();
				auto result = bfs::TSequentialBFS<unsigned>::Do(grid, 0,
lastIndex);
				const auto delay = std::chrono::system_clock::now() - start;
				const auto millis =
std::chrono::duration_cast<std::chrono::milliseconds>(delay).count();
				EXPECT_TRUE(IsPathValid(result.value(), grid));
				WriteToReport(std::format("{{ \"name\": {}, \"size\":
{}, \"milliseconds\": {} }}", "Sequential", size, millis));
```

```
                        return          std::make_tuple(static_cast<double>(millis),
std::move(result.value()));
                  }();
                  for(const auto threadsNum : threadsNums) {
                        const auto start = std::chrono::system_clock::now();
                        const               auto               result              =
bfs::TCommunicationBFS<unsigned>::Do(grid, 0, lastIndex, threadsNum);
                        const auto delay = std::chrono::system_clock::now() - start;
                        const               auto               millis              =
static_cast<double>(std::chrono::duration_cast<std::chrono::milliseconds>(delay).coun
t());
                        EXPECT_TRUE(IsPathValid(result.value(), grid));
                        EXPECT_EQ(result.value(), singleRes);
                        WriteToReport(std::format("{{ \"name\": \"{}\", \"size\": {},
\"threadsNum\": {}, \"milliseconds\": {}, \"acceleration\": {} }}",
                              "Communication",     size,     threadsNum,     millis,
sequentialMillis / millis));
                  }
            }
      }
      }
```